Unambiguous UML (2U) Submission to UML 2 Infrastructure RFP

(ad/00-09-01)

www.2uworks.org

Version 0.8 – June 2002

Submitted by
Adaptive
Data Access
Project Technology
Kinetium
Softlab
Siemens

In association with
Dr A. Clark, pUML group & Kings College, London, UK
Dr A. Evans, pUML group & University of York, UK
Dr S. Kent, pUML group & University of Kent, UK

Supported by
Artisan
Cacheon
J P Morgan Chase
Foundatao
SINTEF
Tata Consultancy Services
University of Kent
Kings College London
University of York

Contents

Preface

0.1 Introduction to the Submission

This is a response to the UML 2.0 Request for Proposals on Infrastructure (ad/00-09-01). We propose an architecture for the definition of UML 2.0 which supports the layered and extensible definition of UML as a ***family*** of languages, and depends on the use of package extension (composition) and package template mechanisms in the metamodelling language. This submission defines that architecture and populates it with the definition of a core foundation for the definition of structural and behavioural modelling constructs for UML. Chapter 3 ("Language Architecture") identifies all those parts of the architecture defined in any given version of this document.

Although this is not a submission to the RFP on the Object Constraint Language (OCL), the definition does include a metamodel for the core of OCL. This is intended to show how OCL can be fitted into our architecture, and we have made every effort to align the metamodel with that proposed in the submission by Boldsoft et al., which the 2U team support. Further alignment may be required in finalisation.

The goal in the revision of UML must, in the end, be to provide better languages and tools to engineers so that they can build better and safer systems, at less cost. This submission aims to deliver on this goal by providing a definition that adheres to seven principles:

1. The definition should be ***unambiguous***, so that questions of understanding, use and conformance can be answered definitively. An unambiguous definition provides a better foundation for provisioning tools.
2. The definition should ***separate concerns***. At one level there should be a clear separation between those aspects of the definition that deal with representation (syntax) and those that deal with the meaning underlying representation. At another level, it is important to identify and separate mechanisms that deal with differing aspects of languages. For example, the mechanism that deals with static information structures (classes) should be separated from the mechanism that deals with behaviour (actions).
3. On the other hand, the definition should support ***integrated modelling languages***. The separate parts of the definition should be formed in such a way that they can be easily combined to form useful languages.
4. The definition should be ***complete***: as far as possible, all aspects of a language (including semantics) should be defined unambiguously. The foundation should be rich enough to support the various modelling paradigms used in UML.
5. The definition should be ***layered and extensible*** to support the construction of new members of the UML family. New modelling languages will require new features. It should be possible to introduce new features on top of existing concepts.
6. The definition should have a ***consistent and disciplined architecture***, so that it can be readily understood and easily extended. For example it should follow well-defined naming disciplines.
7. The definition should be ***checked in a tool***. The size of the definitions warrants it, to be confident that the definition is correct. At a simple level the use of a tool identifies syntax and type errors. However, tools can also be used to validate the definition, by validating the definition against candidate elements of syntax and semantics domain. The tool checking done in this submission is summarised in Section 0.7, "Tool Validation," on page 19.

These principles are in line with the requirements of the RFP and the broader context of the OMG's MDA strategy, which has risen in prominence since the RFPs were issued. A response to the specific requirements of the Requests for Proposals is provided in Section 0.6, "Mapping to RFP Requirements," on page 16. UML has been flagged as one of the key technologies in making the MDA strategy a success. To realise MDA we believe that the definitions of modelling languages in general, and UML in particular, need to be:

- as clear and unambiguous as possible in all aspects (concrete & abstract syntax, semantics), otherwise it will be harder to build tools and training material, and know that these conform to the standard;
- extensible and composable, so that language variants for use in specific application areas can be constructed easily, and so that tools can be configured to support these definitions;
- supported by tools, which means supporting the exploration and validation of models, which are first class artefacts in MDA, not just supporting their syntactic representation.

The definition of UML proposed in this submission meets the first two of these requirements. It should be easier to build tools to support the definition, not least because it has an unambiguous definition of syntax *and* semantics.

Another key technology for MDA is MOF, which is, after all, the language that should be used to define languages, and (after its revision to version 2) should also support the definition of transformations between metamodels, which is critical to the success of MDA. This submission shows how the MOF modelling language can be defined as a UML family member, using the package extension (composition) and template mechansims. Of course, those mechanisms (whose metamodel definition is also provided here) are included in that language, so that MOF can support the extensible and composable definition of languages, as required by MDA. The package extension and template mechanisms provide one embodiment of an approach to aspect-oriented design; they enable us to apply this approach in the *design* of languages.

0.2 OUTLINE OF THIS SUBMISSION

The submission comprises this preface and two parts, each containing a number of chapters.

Preface	An introduction and overview of the document, a description of the submission team, a change history, a statement indicating how the various RFP requirements have been met, an summary of the work done to validate the definition in a tool, and a statement of what it means to conform to the standard.
Approach	Three chapteres: • An overview of the language used to formulate the definitions, including the language itself which is subset of UML and, it is proposed, will be at the heart of MOF. • A description of the overall architecture of the UML family of languages, and the identification of those languages and language units that are defined in this document. A guide on how to read each chapter in the "Definitions" part. • A description of how the approach supports the extension of UML and the definition of Profiles.
Definitions	A series of chapters providing the full metamodel definitions of templates, language units and languages that lays the foundation for the UML family. The definitions are supported by informal descriptions of the language components and illustrated with examples.

0.3 SUBMITTERS AND CONTRIBUTORS

Submitters		
Adaptive	Pete Rivett	pete.rivett@adaptive.com
Data Access	Cory Casanave	cory-c@enterprise-component.com
Kinetium	Desmond D'Souza	desmond@kinetium.com
Project Technology	Steve Mellor	steve@projtech.com
Softlab	Andreas Elting	andreas.elting@softlab.com
Siemens	Ilir Kondo	ilir.kondo@siemens.at
Other Contributors		
King's College and pUML group	Tony Clark	anclark@dcs.kcl.ac.uk
University of York and pUML group	Andy Evans	andye@cs.york.ac.uk
University of Kent and pUML group	Stuart Kent	sjhk@ukc.ac.uk
Tata Consultancy Services funded researchers and consultants	Biju Appukuttan Girish Maskari Laurence Tratt James Willans	biju@dcs.kcl.ac.uk girishmr@cs.york.ac.uk laurie@tratt.net jwillans@cs.york.ac.uk
BAE Systems funded researchers	Paul Sammut	pauls@cs.york.ac.uk

0.4 ACKNOWLEDGEMENTS

We would like to thank many others from various organisations for direct input of ideas, review and comments. Thanks in particular to Steve Cook (IBM) for initiating the feasibility study (Clark et al., 2000), which was an important step towards realising this work. We would also like to acknowledge funding from Tata Consultancy Services and BAE Systems.

0.5 DOCUMENT HISTORY

Nature of submission	Date
Combined Infrastructure and OCL initial submission	August 2001
Superstructure initial submission.	October 2001
UML 2.0 submission (combined Infrastructure, Superstructure and OCL) version 0.51.	December 2001
UML 2.0 submission (combined Infrastructure, Superstructure and OCL), version 0.61	January 2002
UML 2.0 submission (combined Infrastructure, Superstructure and OCL), version 0.75	April 2002
UML 2.0 submission (combined Infrastructure, Superstructure and OCL), version 0.76	April 2002
UML 2.0 revised submission to the Infrastructure RFP version 0.8	June 2002

TABLE 1. Document History

This section maintains a history of revisions to this document (summarised in TABLE 1. above), including the OMG milestones that the document has been submitted to, and outlines significant changes between each revision.

Changes in 0.8

Organisation

- Refocussed document just on Infrastructure RFP. Infrastructure = a core set of templates, language units and languages to support the definition of both structural and behavioural aspects of UML.

Technical

- Completed "Approach" part, specifically filled in gaps in chapters on "Architecture", "Metamodelling Language" and "Language Extension & Profiles".
- Updated "Preface" including a rewrite of the mapping to RFP requirements.
- Added two new chapters on behaviour and messaging. The former is to define a core semantic model for behaviour. The messaging chapter describes an abstract transport mechnism for object communication & its semantics.
- Generally tightened up the chapter structures and cross checking of templates, etc.
- All chapters have at least one object diagram representing a metamodel instance, for validation purposes.
- A model of scope and environment has been added.
- A mapping from package hierarchy to class hierarchy has been added. These rules show how, in principle, a package hirearchy based on package extension can be implemented as a class framework in an OOPL.

Changes in 0.76

- Harmonised format of definitions chapters.

- Added more instances of the metamodel, represented as object diagrams, for illustration.

Changes in 0.75

Organisation

- Re-designed to conform to a more traditional format for standards documents. Specifically, the first part (preamble) has been reduced to a single Preface. Also, there are now just two other parts: A part describing the "Approach", and a part detailing the metamodel "Definitions" themselves.
- Detailed descriptions of the metamodels have now been put back in the submission, in revised form (they were removed in version 0.61, whilst work was being done on reworking and extending the definitions in the MMT tool).

Technical

- Templates and stamped out models now conform more closely to existing UML standards and the work being carried out by other UML 2.0 submittors.
- The coverage of the metamodels is far wider than in any previous version. In particular, it now includes detailed metamodels for expressions, including OCL, and for actions and operations, including semantic primitives for dynamic behaviour.

Changes in 0.61

Organisation

- Detailed descriptions of the metamodels (language units and languages) and templates have been removed from the document. Instead (tool generated) web-based documentation for these can be obtained from www.2uworks.org/documents.html.
- Chapters overviewing the language unit metamodels and templates have been added.

Technical

- The metamodels and templates are now completely defined using a tool. The tool can generate web-based documentation for the models loaded into it.
- The metamodels have been brought in line with the architecture described in this document.
- A definition of (the structural aspects of) OCL has been added. Although not yet fully aligned, it is intended to bring this into alignment with the OCL submission submitted by Boldsoft et al.
- The models of DataTypes and Associations, on the structural side, have been refactored.
- The architecture of the behaviour language units has been worked out, and some of the core parts of this have been filled in, specifically fundamental additions to classes and packages from structure, and actions.
- The templates have been simplified and redundant templates removed. Terminology used in templates has been improved.

Changes in 0.51

Organisation

- The document structure has been overhauled. It is now organised into 6 parts: “A: Preamble”, “B: Architecture and Approach”, “C: Infrastructure”, “D: Language Definition and Extension”, “E: Superstructure”, “F: Backwards Compatibility”, “F: Templates”.
- This chapter and a statement of conformance chapter have been included in the part “A: Preamble”. Also included is a chapter on mapping to RFP requirements (also in “Preamble”), which replaces the “Preface”. The overview chapter has been replaced by two chapters: “Introduction” and “Submitters and Contributors”
- The “Context” chapter has been replaced by a chapter on the architecture of the definition (in Part B). The “Metamodeling Approach” chapter has been renamed “MOF.LDL - Informal Description”, as it provides an

informal description of the language for defining languages, which is used for all definitions throughout the submission.

- The “Templates” chapter has been reorganised and revised into the part “F: Templates”.
- Part C now contains the chapters on “Static Core” and “Dynamic Core”, which have been retitled “Static Infrastructure” and “Dynamic Infrastructure”.
- The parts on “D: Language Definition and Extension” and “F: Backwards Compatibility” have been added as placeholders.

Technical

- The architecture of the definition has been overhauled, to be much clearer about the whole UML family, its relationship to MOF and the parts contributed by this document.
- Templates and packages are currently undergoing major revision, to take into account comments received and to capture more of what is required. These should start to appear in the next release.

0.6 MAPPING TO RFP REQUIREMENTS

0.6.1 General Requirements

Proposals shall enforce a clear separation of concerns between the specification of the metamodel semantics and notation, including precise bi-directional mappings between them.

The initial submission clearly separates semantics and (abstract) syntax. Both are metamodeled, and the mapping between them is also precisely modeled. In principle, concrete syntax and its mapping to abstract syntax can be treated in a similar way, though the submission has not done this.

Proposals shall minimize the impact on users of the current UML 1.x, XMI 1.x and MOF 1.x specifications, and will provide a precise mapping between the current UML 1.x and the UML 2.0 metamodels. Proposals shall ensure that there is a well-defined upgrade path from the XMI DTD for UML 1.x to the XMI DTD for UML 2.0. Wherever changes have adversely impacted backward compatibility with previous specifications, submissions shall provide rationales and change summaries along with their precise mappings.

The architecture supports the metamodeled definition of mappings between metamodels. The submission does not provide these mappings in detail, as (a) they should be done when a metamodel for UML 2.0 has been finalised and (b) could usefully use the results of the MOF transformations RFP to express them. Upgrade from XMI 1.x to XMI 2.0 can be achived via implementations of these metamodel mappings.

Proposals shall identify language elements to be retired from the language for reasons such as being vague, gratuitous, too specific, or not used.

It is only possible to make such a list once both superstructure and infrastructure has been fully defined. This submssion, therefore, refrains from identifying such language elements.

Proposals shall specify an XMI DTD for the UML metamodel.

An XMI DTD (or schema) will be generated from putting the metamodel through the MOF tools. For this, the metamodel has to be in a certain form (Essential MOF – see MOF 2 submission from IBM et al.). In particular, it can not use package extension or templates. The rules provided in Appendix A show how a package extension hiererachy can be converted into a class hierarchy which is suitable input for XMI DTD (or schema) generation by MOF tools. There are also rules (explained informally in Chapter 3) and defined in the metamodel for package extension, which allow a package in a package hierarchy to be expanded so that it is no longer dependent on the hierarchy. These expansions are also in a form suitable for processing by MOF tools, and provide an alternative source for generating XMI DTD/schema.

0.6.2 Architectural alignment and restructuring

Proposals shall specify the UML metamodel in a manner that is strictly aligned with the MOF meta-metamodel by conformance to a 4-layer metamodel architectural pattern. Stated otherwise, every UML metamodel element must be an instance of exactly one MOF meta-metamodel element. If this architectural alignment requires that the MOF meta-metamodel also needs to be changed, then those changes (including changes to XML and IDL mappings) should be fully documented in the proposal.

The metamodels in this submission are defined using a metamodelling language that is summarised in Chapter 3. The metamodeling language is, effectively, a revised form of the MOF 1.4. language, enhanced with package extension and package template facilities. The metamodel for the abstract syntax and semantics of this language is defined in the definitions part of the document. The intention is to ensure that, after finalization, this metamodel matches exactly with the metamodel define in MOF 2 (see below).

The metamodel is an expression in the language for which it is a metamodel of, as are all the other metamodels defined in this submission. Chapter 2 ("Architecture") includes further discussion on the 4-layer metamodel architecture pattern.

Proposals shall strive to share the same metamodel elements between the UML kernel and the MOF kernel, so that there is an isomorphic mapping between MOF meta-metamodel kernel elements and UML metamodel kernel elements.

A new version of MOF is defined in the submission to be a member of the UML family of languages. It shares all its model elements with those of other members of the family that require similar capabilities. In subsequent revision or finalization, the metamodel will be refactored to align with the definition of MOF in the MOF 2.0 submission. This will be achieved by refactoring the package templates so that, under the mapping rules described in Appendix A, the class hierarchy that results from mapping the abstract syntax parts of the metamodel, matches exactly with that defined in the MOF submission (either its MOF or EMOF form).

Proposals shall restructure the UML metamodel to separate kernel language constructs from the standard elements that depend on them. The standard elements shall be restructured consistent with the requirements in 6.5.3.

The architecture described in chapter 2 separates out package templates from language units from languages. These are further categorized into templates/language units that are UML specific and those that could be used to support the definition of languages not in the UML family. This submission, on infrastructure, identifies a core set of language units, with the templates to support their definition. This core, we believe, provides a foundation for defining most structural and behavioural aspects of UML.

Proposals shall decompose the metamodel into a package structure that supports compliance points and efficient implementation.

See the compliance statement at the end of this preface for an indication of how the architecture defined in chapter 3 supports different compliance points. It does so very cleanly.

Appendix A defines a series of mapping rules which provide one route through to implementation: they show how to convert a package hierarchy to a class hierarchy that can then be processed through MOF tools to build and implementation of the metamodel. The definition has also been run through a tool which is able to expand a particular package in the hierarchy so that it is no longer dependent on the hierarchy. The result can also be processed through MOF tools.

Proposals shall identify all semantic variation points in the metamodel.

Our architecture supports the ability to define families of languages which may vary in their semantics in some places and be common in others. If a language does not support quite what is required, then infrastructure support is provided through reusable templates to define an alternative language, or language unit, that can be combined with existing languages or language units. See Chapter 5 on Language Extension and Profiles for more details.

Proposals may refactor the UML metamodel to improve its structure if they can demonstrate that the refactoring will make it easier to implement, maintain or extend.

The submission refactors the UML metamodel somewhat. The refactoring is a direct consequence of defining the semantics in terms of primitives on which the remainder of UML 2.0 is built. This layered approach is easier to implement, maintain and modify.

Proposals may consider architectural alignment with other specification language standards.

Not applicable.

0.6.3 Extensibility

Proposals shall specify how profiles are defined.

Examining so-called profiles currently being standardised in the OMG, one observes that the following approach is adopted: define a new metamodel; show how that metamodel maps into UML notation, specialised

using stereotypes etc. This submission provides two mechanisms – package extension and package templates – that make it much easier to combine and extend fragments of metamodel to form new members of the UML family of languages. This is a accompanied by an architecture which is populated with reusable templates for language design and predefined language units. No longer will profiles need to define a new metamodel from scratch; there is a whole infrastructure on which they can build. Furthermore, as the infrastructure supports the definition of concrete syntax and semantics, both these aspects of a profile definition can be handled in a similar way. This explained further in Chapter 4, which also explains how to mix in a simple definition of stereotypes and tagged values to support a very lightweight form of extension, which is useful for bespoke, user-defined extensions of UML notation, but not recommended for the definition of profiles.

Proposals shall specify a first-class extension mechanism that will allow modelers to add their own metaclasses, which will be instances of MOF meta-metaclasses. This mechanism must be compatible with profiles and consistent with the 4-layer metamodel architecture described in 6.5.2.

The mechanisms used to construct UML profiles are first class extension mechanisms. That is, they work directly with the metamodel.

Proposals shall identify model elements whose detailed semantics preclude specialization in a profile. If proposals need to generalize these model elements, they should propose refactoring consistent with the architecture and restructuring requirements described in 6.5.2.

Not applicable.

Proposals may support the definition of new kinds of diagrams using profiles.

The infrastructure supports the definition of concrete syntax in metamodels. Thus the mechanisms used to build profiles can also be used to add new kinds of diagrams in the definition of a profile.

0.6.4 Issues to be discussed

Proposals should provide guidelines to determine what constructs should be defined in the kernel language and what constructs should be defined in UML profiles and standard model libraries.

This issue is discussed in Chapter 2 on Architecture.

Proposals should stipulate the mechanisms by which compliance to the specification will be determined, recognizing that determination of conformance is on a subset of the specification and that not all parts of a metamodel package are always needed. For example, proposals might submit XMI DTDs to test the compliance of a tool to the specification in a subset of a metamodel package.

See the section on compliance in this preface.

Proposals should discuss the impact of any changes to the UML metamodel on adopted profiles. In particular, the impact of any refactoring should be discussed.

The metamodels of many of the existing profiles are not constructed on top of the UML metamodel at all. It would be advisable to refactor these metamodels to be based on the library of templates and language units defined in this submission, and, where those are found wanting, incrementally extend that library. Of course it would also be advisable to bring those profiles (e.g. SPEM) that are based on the UML metamodel, to be based on the library defined here. Further discussion of our vision of how this infrastructure submission supports the eveolution of the UML *family* of languages (including profiles) is provided in Chapter 4, "Language Extension and Profiles".

0.7 TOOL VALIDATION

In accordance with our seventh principle, the majority of the metamodels defined in this submission have been checked in a prototype tool (MetaModelling Tool – MMT, screenshot provided in Figure 0-1).

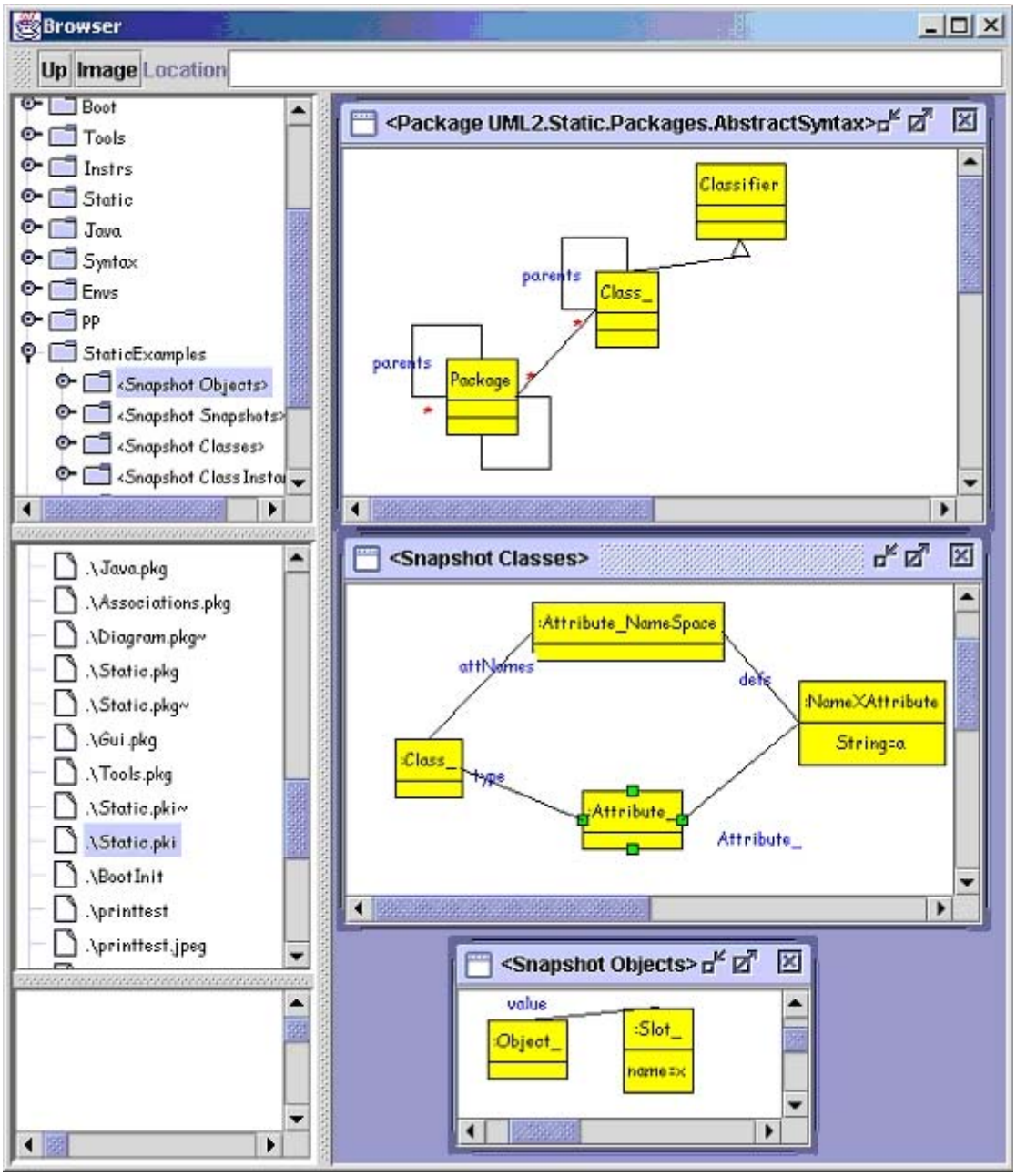


Figure 0-1 *MMT screenshot*

What this means is that definitions have been rendered in a human readable textual notation accepted by the tool which matches, in a fairly transparent way, the graphical definitions presented in this model. All source files are available from the submission website (www.2uworks.org).

The tool is actually being developed to support MDA. The features used to support this submission are:

- Syntax and type checking of all input, including OCL constraints.

- A prototype implementation of the package extension and templates mechanism. This has been used to process package extension and template instantiation hierarchies and generate the expanded form of any package in that hierarchy. The tool is not currently able to automatically generate documentation of these expansions (this is a resource problem, not an inherent limitation of the tool), but is able to generate some useful elements of the expansion in text form (e.g. constraints) which have been pasted into this document.
- Construction of instances of the metamodels, and checking that they satisfy all well-formedness constrraints on the metamodel. A number of these have been constructed to provide some validation that the metamodel presented in this document captures the required concepts.

0.8 COMPLIANCE

Overview

The architecture of this submission disguishes language units from languages, where a language is a particular combination of language units. Each language unit / language has three components: concrete syntax(es), abstract syntax, semantics domain, with requisite mappings between them.

Thus a statement of compliance should be clear about which languages and language units the tool or method supports, where a language is a particular combination of language units. It should also be clear as to what aspects of a language or language unit definition it supports: concrete syntax and/or abstract syntax and/or semantics.

XMI for a language or language unit can be thought of as an XML concrete syntax (the interchange syntax) for a metamodel. MOF XMI tools support the generation and implementation of this syntax for any MOF-compliant metamodel in a standard way, where 'implementation' means the generation of a parser and generator for the metamodel specific XMI.

A useful way of presenting a compliance statement is to use a table, which lists language units and/or languages as rows, and aspects of the definition of a language or language unit as columns, one each for asbtract syntax and semantics, and for each concrete syntax (including XMI). Compliance to a language that is the combination of a number of language units automatically guarantees compliance to those language units.

Finally, if a new language or language unit is constructed and has not been ratified by the OMG, then one can not claim it to be a member of the UML family, and complying to that language or language unit can not be a claim to compliance with UML. On the other hand, it may still be possible to claim compliance to any language or language unit, that is part of the UML family and which is extended by the new language or language unit.

Test Examples

Compliance should be checked through a representative sample of example models which are expressed using the language or language unit in question. Some examples will not be well-formed. Some examples will come in pairs, where the second in the pair will be like the first except for a designated set of changes. The examples may be provided in a number of formats: as instances of any of the concrete syntaxes defined for that language or language unit (including XMI), or as instances of the abstract syntax metamodel. Semantic compliance will also need example abstract syntax / semantic domian pairs. Some examples have been provided in this submission to validate the definitions (see chapters in 'Definitions' part of document). This set will need to be expanded during finalization.

Concrete Syntax (including XMI)

A tool claiming concrete syntactic compliance to a language or language unit must demonstrate its ability to read in the example in appropriate forms (e.g. if it claims compliance to XMI, then it should be able to read in the XMI), and provide some way of notifying or enforcing well-formedness. To demonstrate that it can output files appropriately, it will read in an example from a pair, the designated changes will be perfomed in the tool, and the example will then be output and checked against the second element of the pair. Some of this process can be automated.

Abstract Syntax

A tool claiming compliance to abstract syntax should provide a standard API (e.g. JMI or IDL) to its model repository.

Semantics

A tool claiming semantic compliance must provide an ability (e.g. through XMI or a JMI compliant API) to access elements if the semantic domain. It should demonstrate that it is able to determine whether or not a semantic domain element is well-formed, and whether or not it satisfies an expression of abtract syntax. This could be tested using a standard set of AS/SD pairs.

Automatic Compliance Testing

Automatic compliance testing will only be possible for tools that support a standard API, which could be, for example, JMI or IDL generated from the metamodel definition of any aspect of the language or language unit definition. Then test scripts can be written which automatically feed in examples to the tool and check results. It may be possible to automate testing for tools that support XMI, but then some specification of how the input and ouput of XMI is orchestrated will be required.

APPROACH

Chapter 1 Introduction

This document provides a definition of UML 2.0. It has two main parts.

The *first* part (*Approach*) includes:

- This *Introduction*
- A description of the *Metamodelling Language* in which the definition is unambiguously expressed
- A description of the overall *Architecture* of the definition, expressed in the metamodelling language
- A description of how the approach supports the *Extension* of UML, including the definition of UML *Profiles*

The *second* part (*Definitions*) provides a series of chapters detailing, explaining and illustrating the definitions. Chapter 3 ("Language Architecture") in the *Approach* part, provides an overview of the content of these chapters. The metamodel definitions are interspersed with chapters explaining and illustrating parts of the definition through examples.

A reader who wishes just to understand UML in an informal way, should begin by reading the example chapters in the Definitions part. A reader who wishes to gain a formal understanding of the definition, in order to build a tool, for example, should begin by reading the *Approach* part (at least the *Metamodelling* and *Architecture* chapters) before the *Definitions* part.

A metamodelling approach is used for the definition of UML. In essence, this means the definition of syntax and semantics as object models. The metamodelling language is an object modelling language that is a subset of UML itself (hence defined in this document). This risks circularity in definition, which can be broken in a number of ways:

- The metamodelling language is small enough and uses commonly enough used concepts that one can be confident in understanding what it means intuitively. Any questions can usually be answered by looking closely at the definition of itself in itself.
- The metamodelling language is implemented in a tool, which validates the syntax and well-formedness of definitions, and provides a means to validate the language semantically.
- The metamodelling language is defined in another formalism (e.g. mathematical set theory), which may may be used to increase one's confidence that it is correct and captures the desired concepts.

The definition of the UML 2.0 infrastructure provided by this document uses the first two devices to break circularity. In particular, a tool implementation of the metamodelling language has been favoured over a mathematical definition, as not only does it provide a similar level of confidence in the definition, it also provides a useful tool for validating metamodels, including the definition of itself in itself!

The definition of UML infrastructure is architected in a way that directly supports the notion that UML is a family of languages, not a single language. Thus *Language Units* are defined, each focussing on a particular grouping of language features (e.g. model management and packaging, structural modelling, constraints, various forms of behavioural modelling, etc.). Language units can be composed to form different *Languages*. Both language units and languages are constructed from language definition templates, which, amongst other advantages, help to enforce a consistent architecture across the definition. The definition architecture is described in Chapter 3 ("Language Architecture").

Templates also help in the extension and construction of new language units, which can then, in turn, be composed with existing language units (possibly) to form new languages. This process is explained in Chapter 4

("Language Extension and Profiles"), which also provides guidelines for determining the status of these new language units and languages, with regard to UML 2. The default is that they are not part of the UML 2 family, though, of course, languages and language units developed in this way may be standardised as official UML profiles using normal OMG procedures.

Chapter 2
Metamodeling Language

This chapter provides an informal description of the language used to define the UML 2 metamodels. The language used is itself a member of the UML family of languages, so is defined in itself as part of this document (see Chapter XX), and is the language for metamodeling employed by OMG's meta-object facility [**Note**: Or so it is proposed by this submission].The metamodeling language has the following components:

Classes, attributes, query operations. With associations, provides the means for defining the (unconstrained) structure of all aspects of a language.

Associations. With classes etc., provides the means for defining the (unconstrained) structure of all aspects of a language.

Packages, including **nesting.** Allows related concepts to be grouped into different namespaces. Nesting of namespaces is permitted.

A constraint language (OCL). For expressing well-formedness constraints on the structures admitted by the metamodel.

Package extension and package imports. Provides a means of building packages up incrementally, and a means for composing packages by merging elements within those packages. Can be used to define languages by composing separately defined language components. Package imports is just a (very) restricted form of package extension.

Package templates. Can be used to capture metamodelling patterns in a precise and effective way. Models can be constructed by instantiating one or more package templates, then merging and (optionally) extending the result. The package template mechanism is defined as a layer on top of package extension, which supports the merging or composition of multiple instantiantions from templates, and construction of templates through template composition.

Package extension provides a means for separating out different concerns of a metamodel into separate packages, that can then be merged or weaved together as necessary. Package imports does not include a concept of merging, which makes it much harder to separate out often overlapping and cross-cutting concerns. Package templates provides a simple layer on top of package extension which allows common applications of extension from the package, involving a set of renamings, to be generated from a small number of template parameters.

All six components are described in the remainder of this chapter. Well-know concepts are treated in summary; package extension and package templates are considered in more detail. A metamodel definition of all components is provided in the *Definitions* part of this document.

2.1 Classes, Attributes, Query Operations

Classes, attributes and query operations are permitted in the metamodelling language. Visibility annotations on any of these are not permitted (or should be ignored). Query operations are operations which return a result and

may have arguments. Query operations may be accompanied by an OCL expression whose type is conformant with the type of the result of the operation.

Classes may be specialised. Attributes and query operations may not be redefined, but additional OCL constraints can be used e.g. to stregthen result types.

2.2 ASSOCIATIONS

Binary associations only are included in the metamodelling language. Association classes and qualified associations are not included. Association specialisation is not permitted.

2.3 PACKAGES

All classes and associations must be defined in the context of a package. Packages may contain other packages, so a package may contain a mixture of classes, associations and packages. There are no constraints on the types of associations ends, attributes and parameters/result of queries, with respect to packages. For example, it is not necessary for the type of an attribute to belong to the same package as the class in which that attribute is contained. This does mean, however, that some cases can be difficult to represent graphically, for example if there is an association contained in Package P, whose ends refer to classes in package Q.

Similarly a class C may specialise classes from packages which do not contain C.

2.4 CONSTRAINT LANGUAGE

The metamodelling language uses the object constraint language (see submission to the UML 2.0 OCL RFP) to express invariant constraints on classes, and for expressions that determine the value returned by a query operation (in such cases the type of the expression must conform to the return type of the query operation). An example of the latter is provided below:

```
context Class::conformsTo(c : Class):Boolean
  self.generalElements()->includes(c) or self = c
```

This defines a query operation *conformsTo* on the class *Class*, whose result is calculated by evuating the OCL expression appearing on the second line.

2.5 PACKAGE EXTENSION & IMPORTS

We describe package extension and package imports together, as package imports is really just a restricted form of package extension. The restrictions are so draconian, that, in practice, package extension tends to be used.

2.5.1 Package Extension

Package extension provides two facilities to the metamodeller:

- It can be used to extend a fragment of metamodel as a whole, rather then piecemeal (e.g. class by class). This supports incremental definition of language fragments, where each increment may add new features to a number of classes used to define the original fragment.
- It can be used to compose fragments of a metamodel. Here it differs from package imports in the case where a child package is importing two or more packages. Specifically, it merges elements of the parents to form the child, wherever there is overlap between the packages being imported.

The package extension mechanism is illustrated by Figure 2-1.

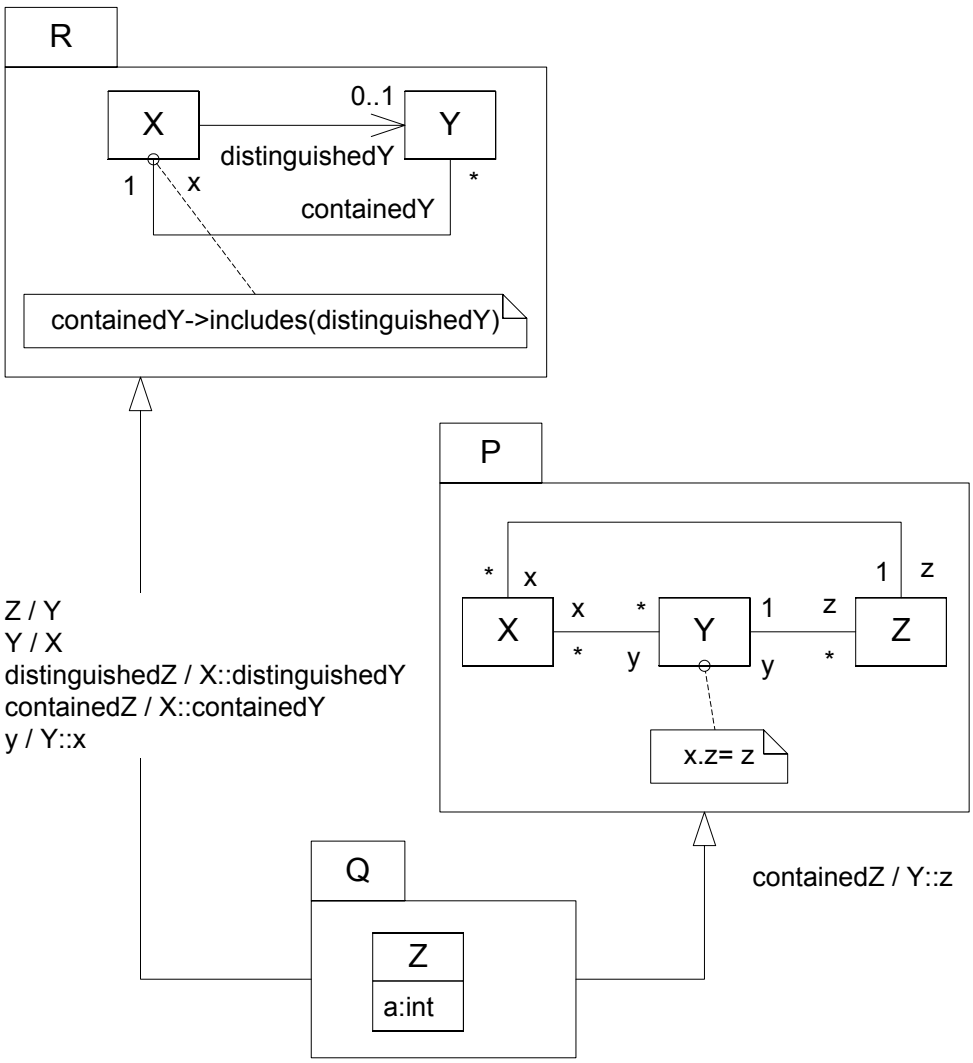


Figure 2-1 *Package Extension*

Q is a package that extends *R* and *P*. Extension between packages is shown by a UML generalisation arrow. The contents of *R* and *P* get included in *Q*, with anything common between the two being merged. Common model elements are elements of the same kind with the same name. Renaming clauses may be used to annotate a package extension either to prevent a merge or to force one. In this case, the classes *X* and *Y* in *R* are renamed to *Y* and *Z*, respectively, to force them to be merged with the classes *Y* and *Z* in *P*. *Q* also contains a fragment a class *Z*, with an attribute *a*, that is also merged with *P::Z* and *R::Y* (which is renamed to *Z*). The unfolding of both package extensions results in the expansion of *Q* which is given in Figure 2-2.

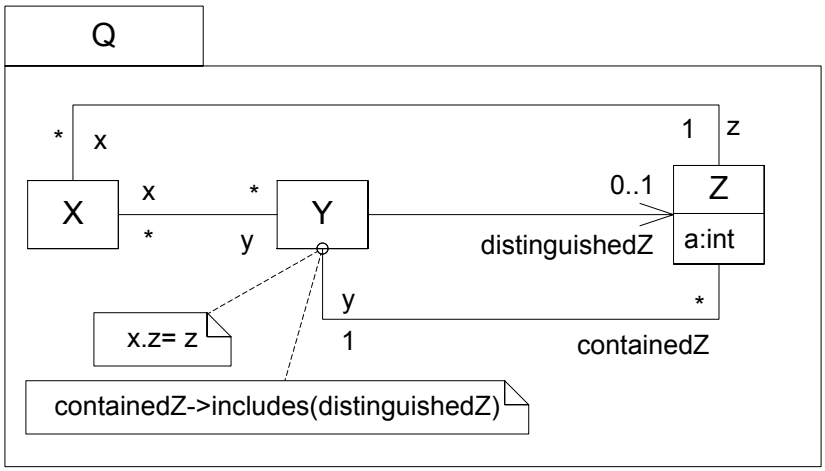


Figure 2-2 *Package Expansion*

As with classes may specialise classes from other packages, so packages may extend packages contained in other packages.

2.5.2 Package Imports

Package imports is a restricted form of package extension. The restrictions are:

- Nothing can be renamed on import.
- The elements being imported can not be merged in the child with elements obtained via import or extension from another package, or elements introduced in the child itself.

These restrictions make package imports easier to define, e.g. in a metamodel, than package extension, but at the severe cost of a considerably weaker notion than package extension.

2.6 PACKAGE TEMPLATES

Package templates allow a package definition to be parameterised over arguments, thereby supporting the encoding of common patterns which can be bound to particular fragments of metamodel through parameter substitution. The package template mechanism is illustrated by Figure 2-3.

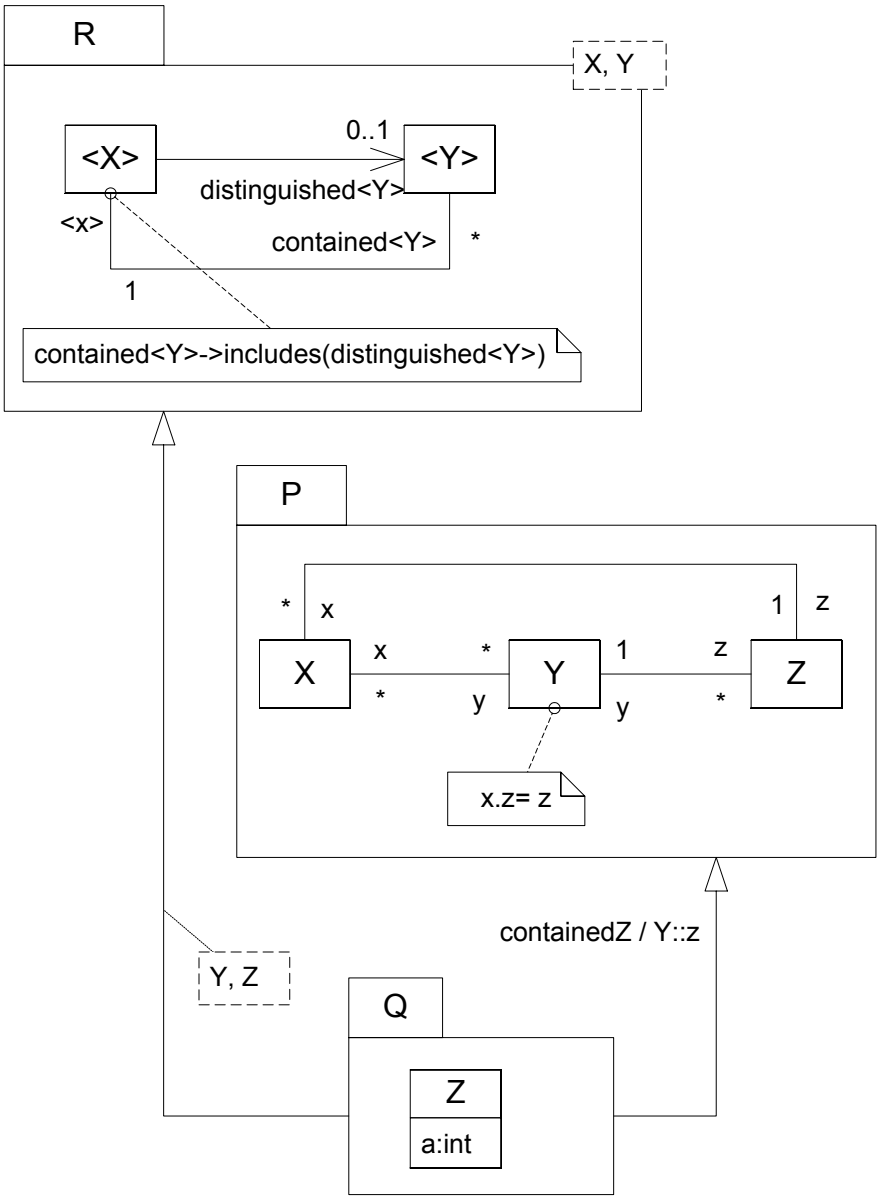


Figure 2-3 *Package Templates*

This is similar to the package extension example of Figure 2-1, except that now package *R* has been turned into a package template. The template takes two string arguments (*X* and *Y* in the dashed box), and names of elements in the package are parameterised by these arguments. Not only are the names of classes parameterised, but also the labels on the association ends, which are referred to in the accompanying constraint.

Instantiation of a template is shown using a generalisation arrow, which must be annotated by a substitution for the arguments, shown by a dashed box called out from the arrow. Template instantiation works by evaluating the

expressions that provide the names for elements in the template with arguments substituted. The result is then merged with the target of the instantiation. A template instatiation may be annotated further with one or more renaming clauses, which override any names calculated from the argument substitutions. In this example there are no such renamings.

Templates effectively allow a (sometimes large) set of renamings to be calculated from a small number of arguments. In this example, the five renamings on the extension from *R* to *Q* in Figure 2-1 are replaced by a substitution for two arguments. Not only does this save work for the modeller, it also ensures more accurate use of the template by forcing a particular set of renamings (which may be overridden in extremis) whenever the template is applied.

Chapter 3
Language Architecture

This chapter defines the overall architecture of the definition of UML 2. The definition is organised into a number of packages related by nesting, imports and package extension. A distinction is made between *language units*, and *languages* composed from these units. Both language units and languages are defined as packages in a layered fashion. Both languages and language units have the same internal architecture, which separates concrete syntax from abstract syntax from semantics. The relationship of MOF with UML is clarified. Backwards compatibility with UML 1.4. is defined using “mapping” packages. The relationship of this approach with the 4-layer model for language definition is explained.

3.1 THE ARCHITECTURE OF UML 2

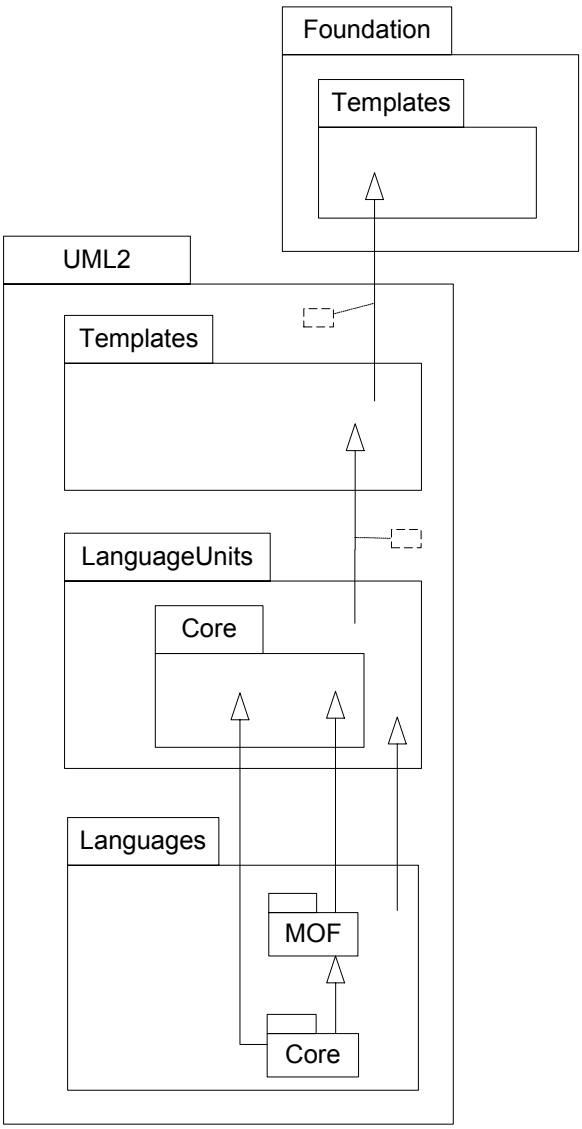


Figure 3-1 *Overall Architecture*

The overall architecture of the definition UML 2 is given by Figure 3-1.

Family of Languages

UML2 is defined to be a family of languages not a single language. This reflects the history of use of UML, where modellers tend to use only a subset of the language (and sometimes a specialised subset) for particular purposes.

Languages and Language Units

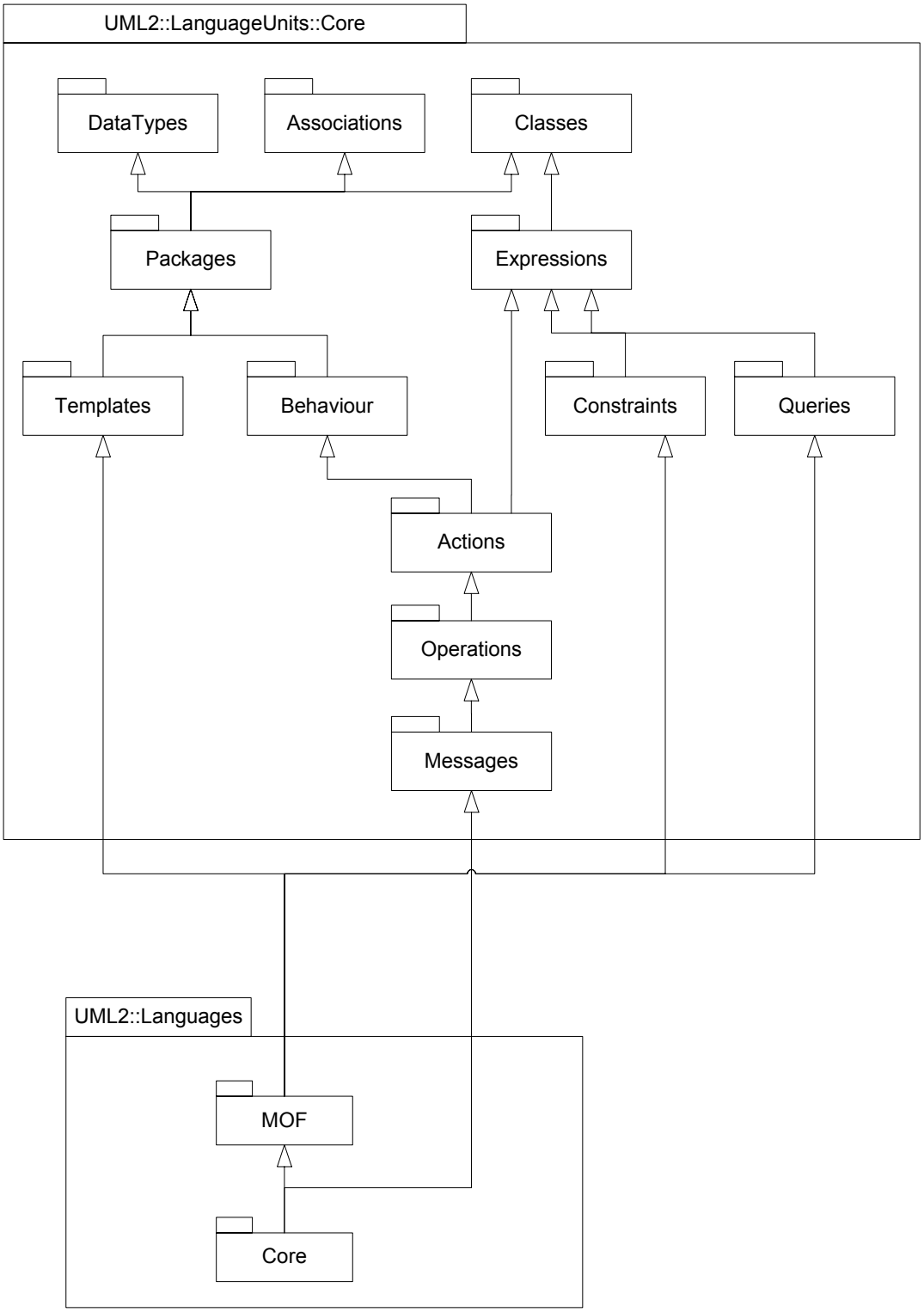


Figure 3-2 *Languages and Language Units*

To support the definition of different family members, the architecture supports the definition of *language units* and *languages*. Language units allow related features of UML to be grouped into separate fragments; fragments may be common to many languages in the family. Language units can be composed, using package extension, to

form complete languages. Package mports is not sufficient in many cases to compose language units, as language units may overlap in content. Package extension allows language units to be merged. The mechanism may also be used to incrementally extend language units.

The distinction between language units and languages is somewhat fuzzy. Although many language units will be mini-languages in their own right, it is expected that they only become practically useful when combined with other units to form a language. Thus the languages are combinations of language units that the designer of the language family has deemed fit for a particular purpose.

There is a core set of language units (a statement of what we mean by core is given below), from which a core language and the MOF modelling language are derived. An overview of the language units and languages defined for UML2 is provided by Figure 3-2. They are detailed and explained in the *Definitions* part of this document.

Core

A subset of the language units have been wrapped in a package called *Core*. This section explains what is meant by "core".

Given the definition of a language syntax (such as UML) there are often a number of design choices to be made regarding the semantic model. It is usual to apply the following principles to the design of a semantic domain: every syntax element denotes exactly one configuration of semantic elements; no configuration of semantic elements can be the denotation of more than one syntactic element.

A consequence of the semantic domain design principles is that the semantic domain should not contain equivalences; i.e. all semantic elements denote distinct concepts. However, for practical reasons it can be useful to define equivalence relationships over a semantic domain: if the domain is used to define a data repository; in order to support an inter-operable tool suite; or, just for conceptual clarity. To meet such practical considerations it is useful to define new semantic elements that represent configurations of existing semantic elements; the new elements do not represent an extension to the expressiveness of the domain, they are provided for convenience.

Given a language definition L it is possible to identify one or more core languages. A core language C of L consists of models of syntax and semantics and a mapping between them such that the extensions added to C to produce L do not extend the expressiveness of C.

The core of UML is defined to a set of language units which together will be expressive enough to support predicted structural and behavioural modelling needs. Together they provide a semantic domain including objects, snapshots and filmstrips, and a syntax domain containing just those features needed to denote elements of the core semantic domain. The core represents the essential features of the UML infrastructure. The UML superstructure does not represent an extension to infrastructure expressiveness and can therefore (in principle) be translated to elements of the core. It is expected that any language in the UML family will, in principle, be translateable into this core. If it turns out that the core needs to be extended to support a proposed new family member, then that will require a revision to the UML core – this should be a consideration whenever a new profile for UML is proposed.

Templates

Package templates are used to capture cross-cutting architectural patterns, and which support the imposition of a uniform and consistent architecture across definitions. The latter is essential for the composition of language units to work correctly. They also ensure more complete definitions by enabling reuse: important structures and constraints are captured once in a template and reused many times over in stamping out definitions of language units. In this way, one is able to reap the rewards from effort invested in a template.

Two groupings of templates have been identifed. Templates which may be regarded as fundamental to language definition per se, capturing concepts such as namespace and typing, and templates which are more specific to UML, using, for example, UML-specific terminology. The UML-specific templates are constructed from the fundamental templates. An overview of the templates used to construct the UML is provided by Figure 3-3, which shows the templates used to build the abstract syntax, Figures 3-4 & 3-5, which showss the templates used to build the semantics, and Figure 3-6 which shows the templates for defining package extension and templates. They are detailed and explained in *Definitions* part of this document.

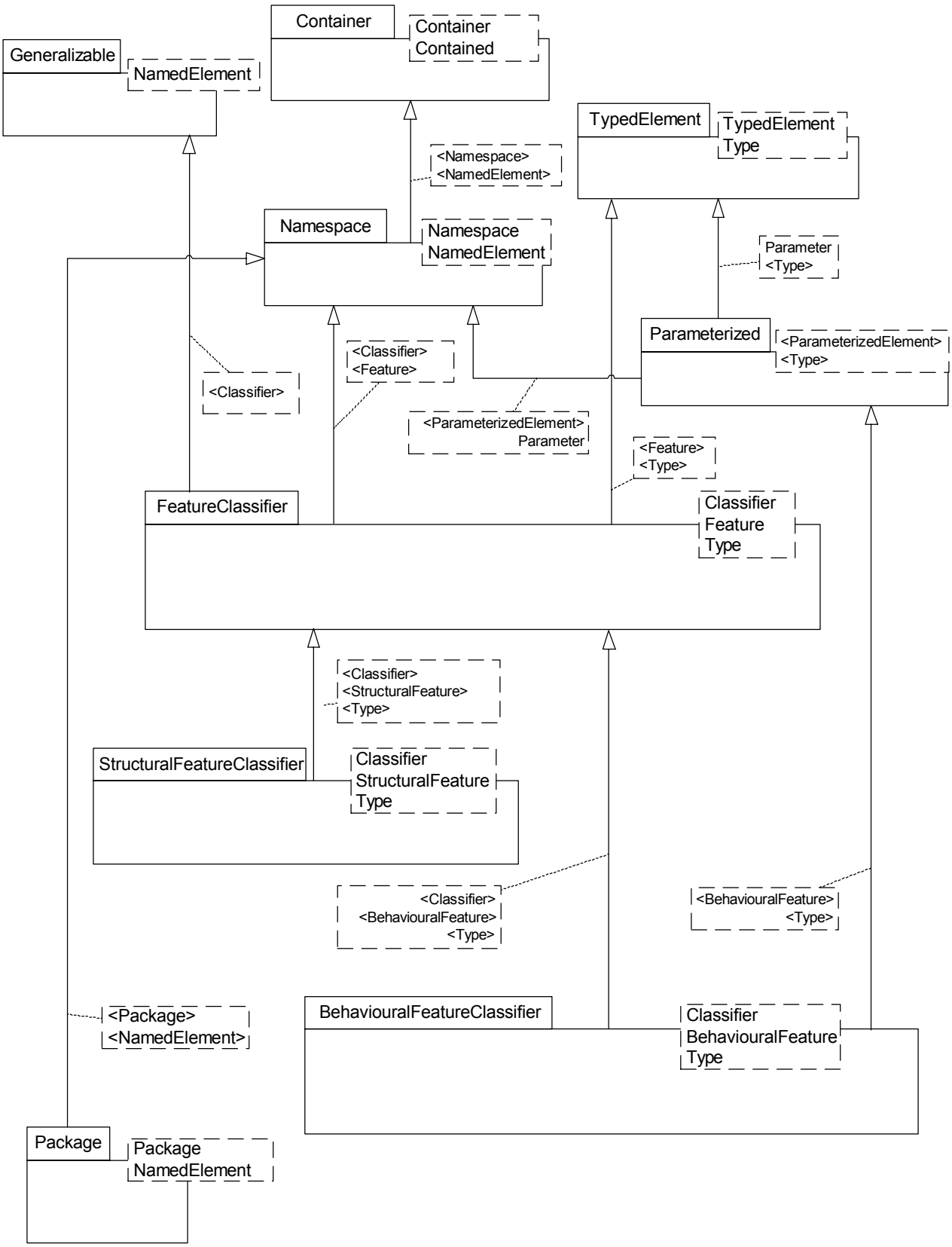


Figure 3-3 *Templates (abstract syntax)*

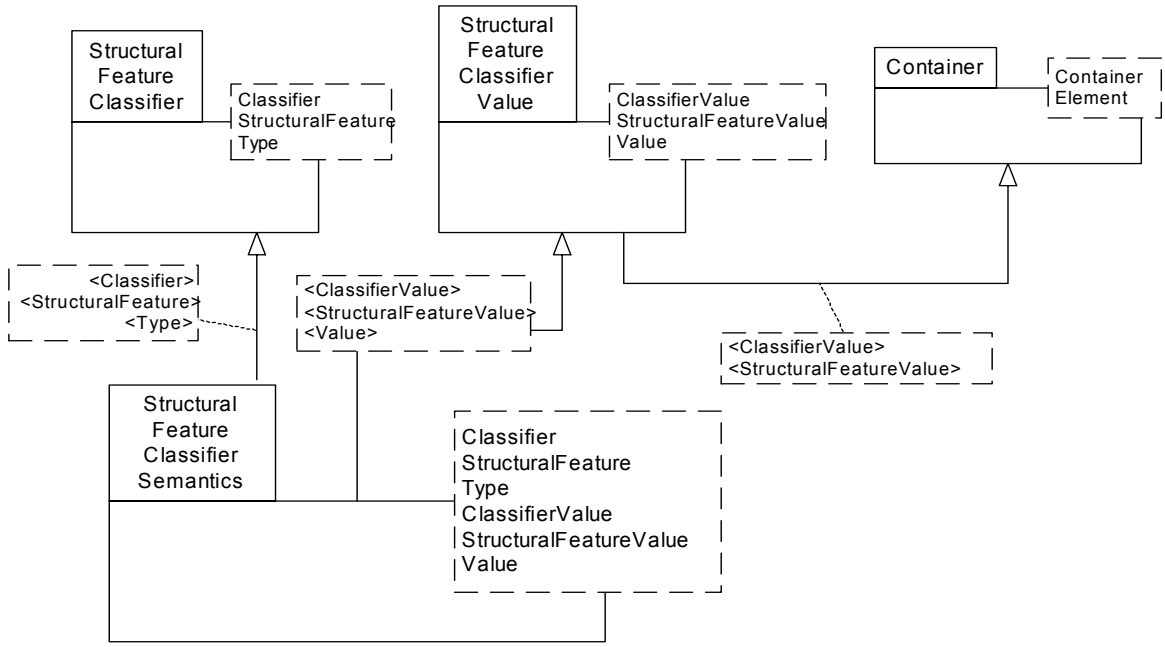


Figure 3-4 *Templates (structural semantics)*

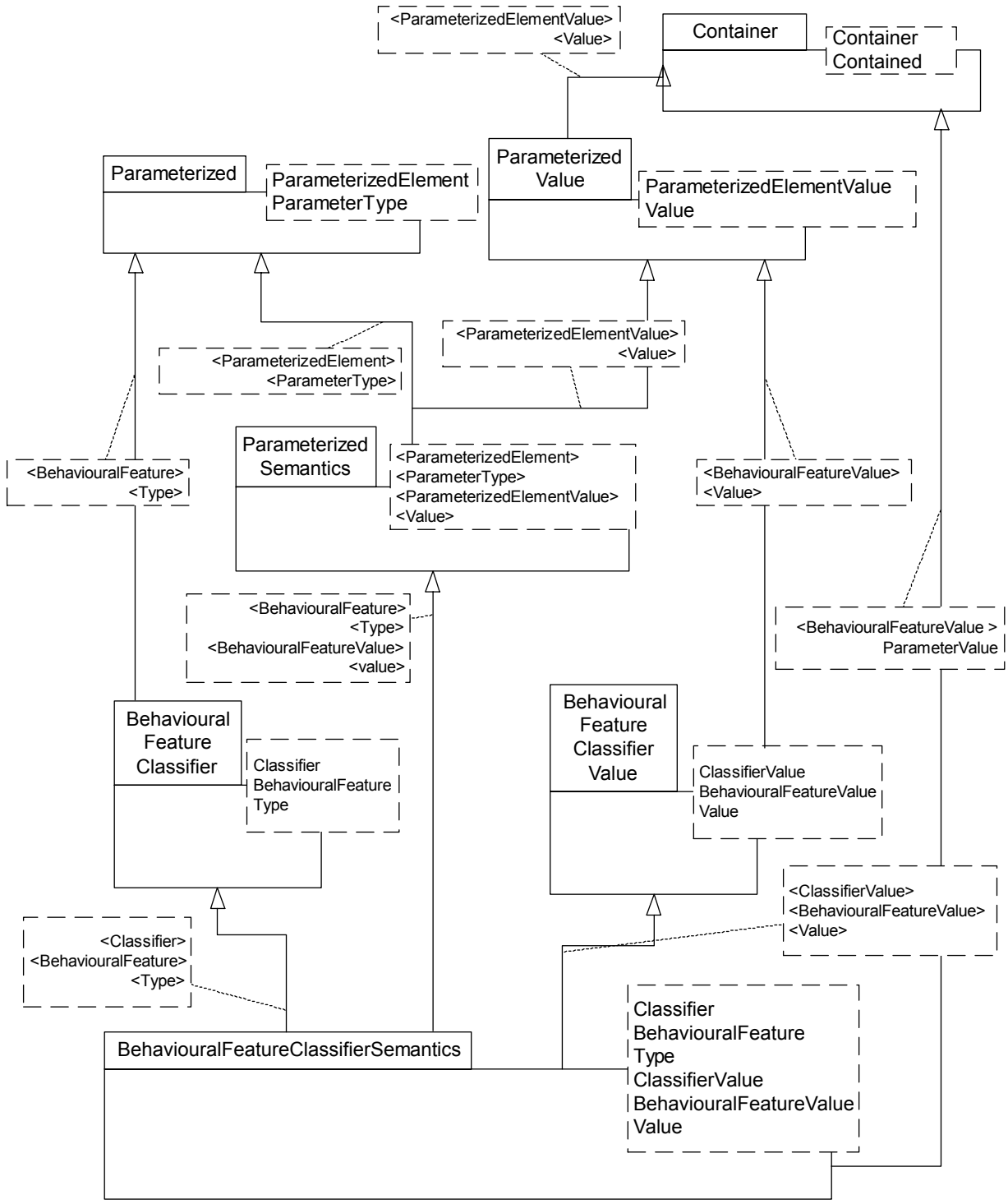


Figure 3-5 *Templates (behavioural semantics)*

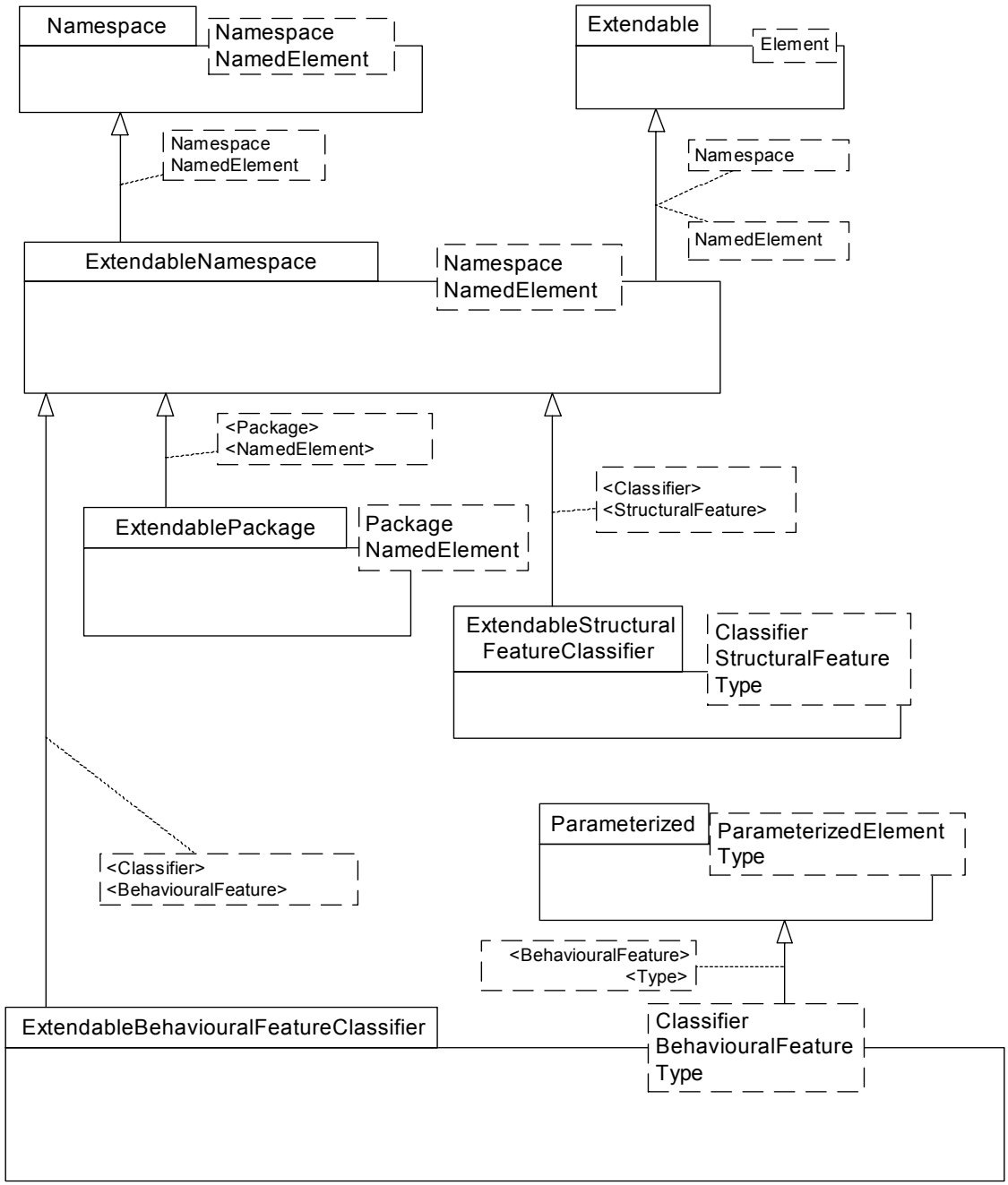


Figure 3-6 *Templates (package extension & templates)*

Syntax and Semantics

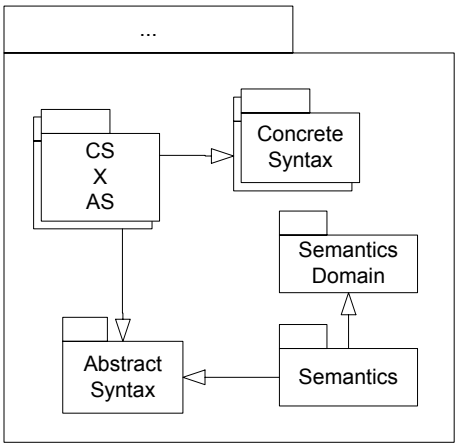


Figure 3-7 *Syntax and Semantics*

The internal architecture of languages and language units is given by Figure 3-7. A language definition comprises any number of concrete syntaxes, an abstract syntax and a semantics domain. The abstract syntax is a model of the valid expressions of the language, abstracted away from from any particular concrete rendition of those expressions. There may be many concrete syntaxes for one abstract syntax. For example, XMI defines how a UML model may be rendered as XML, a concrete syntax. A class diagram is concrete syntax for models constructed from classes and associations.

Semantics concerns the definition of what it means for an example or instance of behaviour to satisfy the specification of that behaviour, as characterised by an expression of the language under consideration. For example, the semantics of a Java program can be given by stating the rules by which an execution trace satisfies aan expression of Java. Because a Java program is deterministic, one might also give the semantics in a slightly different way, that is given a valid starting state, what is the execution trace that is then generated. In the architecture examples of behaviour are defined in the semantics domain. Semantics is then defined to be a mapping between semantics domain and abstract syntax.

Note that semantics in this sense should be distinguished from *static semantics*, which are the rules which dictate whether or not an expression of the language is *well-formed*. Static semantics rules are those employed by tools such as type checkers, and correspond to OCL constraints over the concrete and abstract syntax parts of a language (unit) metamodel.

3.2 MOF

One of the languages in the UML family is the language used in the Meta Object Facility (MOF) for metamodelling. This is defined as a member of the UML family of languages to be the composition of the core language units concerned with structural modelling, including the object constraint language. The construction has already

been given by Figure 3-2. Figure 3-8 illustrates the relationship between this language and other parts of MOF, though these are outside the scope of UML.

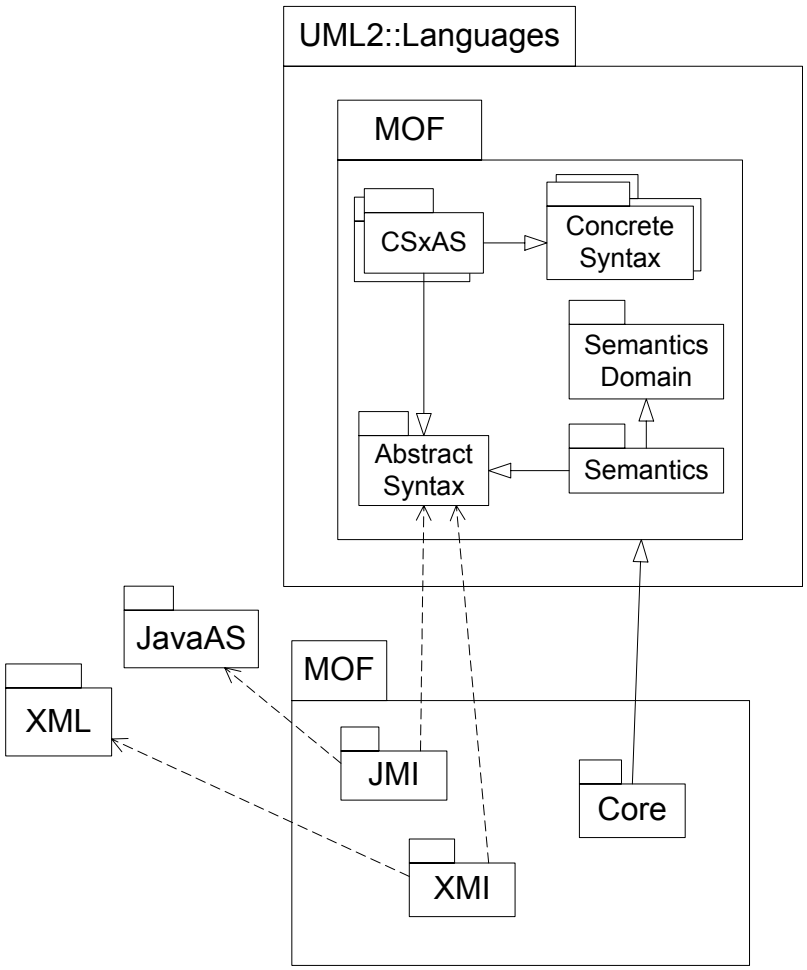


Figure 3-8 *UML and MOF*

Here dashed arrows indicate a package dependency (something inside the source package is dependent on something inside the target package), not package imports, and presumes that JMI (XMI) are models characterising transformations between JavaAS (XML) and UML2::Languages::MOF::AbstractSyntax, in a way that does not intefere with either side of the mapping. If this is not possible, then the dashed arrows would need to be replaced by package extension relationships.

3.3 PROGRAMMING IN PICTURES

Although not explicitly called out in the UML 2 RFP, there is a large community of UML users who tailor UML so that it can be used to provide diagrammatic views of object-oriented programs in specific programming langauges. This can be accomodated by defining a programming language specific UML language, which brings together and specialises the required UML language units. The definition for Java is illustrated in 3-9 on page 43. This captures most current uses of UML profiles for programming languages, which only require UML views of Java programs, not execution traces. For this reason a semantics domain has not been included in the Java profile.

However, the profile must include a definition of the mapping from abstract syntax in the profile to the abstract syntax of Java. Of course the definition of Java is outside the scope of UML.

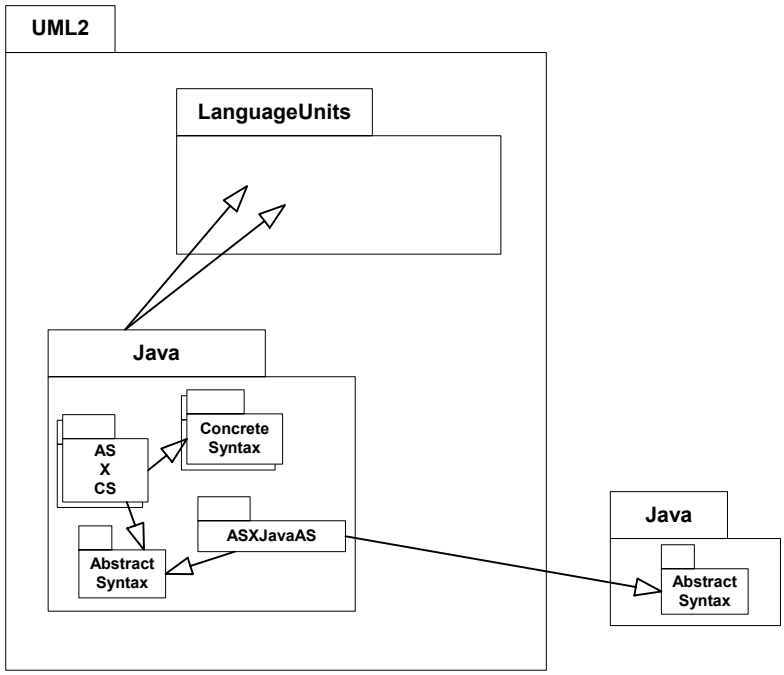


Figure 3-9 *Java profile*

3.4 BACKWARDS COMPATIBILITY

The differences between UML 1.4 and UML 2 can be defined by modelling the mapping between packages in UML 1.4. and packages in the new version of UML, as illustrated by Figure 3-10 on page 44. Not only should

this approach formally define the differences between the two versions, it also provides a specification for tools that will automate the transition.

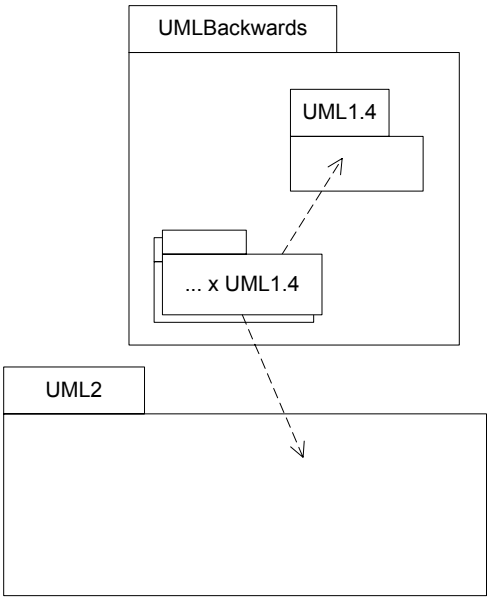


Figure 3-10 *Backwards compatibility*

[**Note**: The details of this mapping should be deferred until it is known what the UML 2 metamodel has been agreed by the OMG. It is not technically difficult to write (it can be expressed as an object model with OCL constraints), just laborious. It might be more appropriate to construct it using MOF technology for transformations; however, that might be in place too late.]

3.5 METALAYERS

In this submission, a metamodel (which is just a model expressed in a particular language) is used to capture and define the relationship between two metalayers: the relationship between models of a particular language, and instances of the models of that language. The abstract syntax metamodel defines the collection models that can be expressed in the language, and the semantics domain metamodel defines the collection of instances of models for that language. The semantics metamodel defines the relationship between the two. This means that a repository generated from the metamodels used in this submission, could store both models and instances of those models, and, if all the well-formedness rules were implemented as checks on the repository, one could check which instances were valid instances of the models.

Of course, a metamodel is itself a model of the language used to express metamodels, and one of those models can be a definition of the metamodelling language itself. This fact allows metamodels to be cast as instances of that model, which can be useful e.g. to check the well-formedness of metamodels, and can be used to support reflection. However, this is enteriing the domain of MOF and is really beyond the scope of a UML submission.

Chapter 4
Language Extension and Profiles

The package extension and package template mechanisms support the definition of a new language based on the UML metamodel, as follows.

- Identify appropriate language units. It may be that the new language can be formed through the composition of the existing language units. In which case, defining the language is a matter of having the new language extend each of the chosen language units.
- Specialise existing language units. The language may require some specialisation of language units before they are composed. For example, it may have stronger well-formedness constraints, or specialist forms of certain model elements. In this case, those units should be extended, and the extended versions composed with any other units required to form the language.
- Create new language units. If there are elements of the language which can not be supplied by existing language units, then it will be necessary to construct new language units. These could be created from scratch, or existing templates used to generate the new unit. The application of templates will depend on the richness and flexibility of the template library. In extremis it may be necessary to define new templates.

The following two questions remain to be answered:

- Is a new language unit or language constructed in this way a member of the UML family?
- Can these techniques be used to support so-called "lightweight" extension, or is something else required?

The answer to the first question is closely related to what is meant by "compliance" to the UML standard. In this submission, we propose that a statement of compliance should be clear about which languages and language units the tool or method supports, where a language is a particular combination of language units. It should also be clear as to what aspects of a language or language unit definition it supports: concrete syntax and/or abstract syntax and/or semantics. Thus if a new language or language unit is constructed and has not been ratified by the OMG, then one can not claim it to be a member of the UML family, and complying to that language or language unit can not be a claim to compliance with UML. On the other hand, it may still be possible to claim compliance to any language or language unit, that is part of the UML family and which is extended by the new language or language unit.

In answer to the second question, the position taken by this submission is that for significant extensions of UML, such as many of the *profiles* currently being standardised within the OMG, the extension should be made directly to the metamodel as described above. Indeed, we note that these so-called profiles are accompanied by new metamodels, often not even based on the UML metamodel, and that the lightweight extension mechanism in UML 1.4. is really only used to tailor the concrete syntax of UML to provide a concrete syntax for these metamodels: there is a mapping from the new metamodel to a specialised UML concrete syntax. If concrete syntax is also defined as a metamodel (a vision of this submission, but not directly tackled by the submission), then the metamodel extension process described above can be used to specialise the concrete syntax in a profile, in conjunction with the abstract syntax, for which new metamodels are already being constructed. Note that the specialised concrete syntax could be, for example, the insertion of the label <<Y>> in the symbol corresponding to model element X, where Y extends X in the metamodel for the profile. Indeed a metamodel template could be written to make such a definition easy.

When a profile is standardised, not only will the new language be standardised, but also any additional templates and language units required to support that profile. Also, issues might be raised against existing units and templates, and an impact analysis can be conducted on the existing supported languages to ascertain the best way

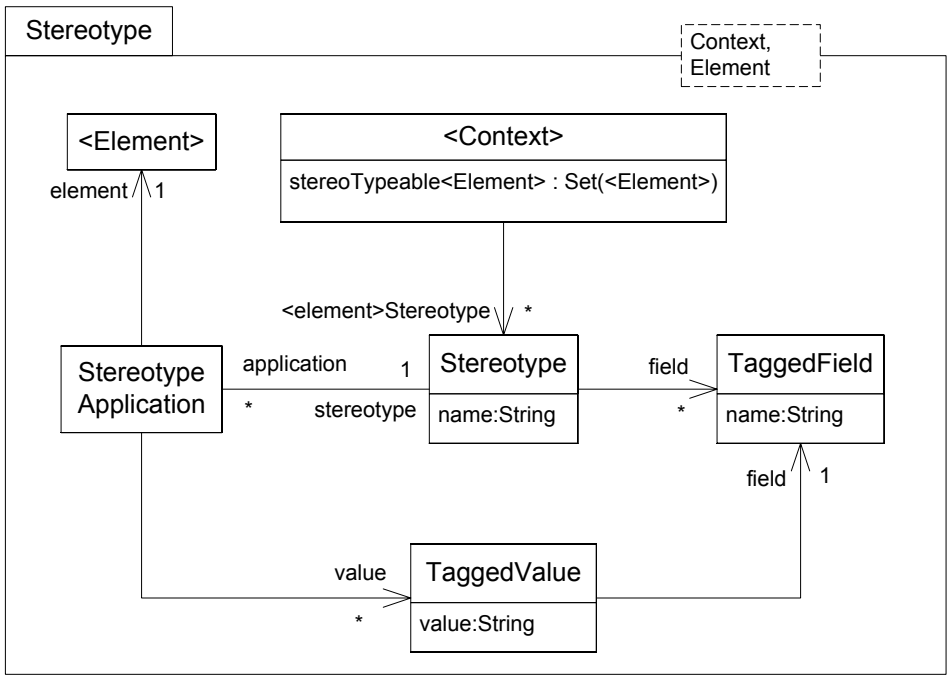
to handle any issue. In this way, the set of modelling languages, language units and templates supported by the OMG can be evolved and expanded in an incremental fashion, avoiding major revisions such as we have now.

The degree to which a tool supports the manipulation of the metamodel, is really a question about the degree to which it supports MOF, not UML.

Nevertheless, this submission does already provide a lightweight extension mechanism. The package extension and template mechanisms defined in the submission can be used for application modelling (standard UML modelling) as well as for metamodelling. They provide a way of capturing standard modelling patterns, and reusing those patterns. Thus it would be possible to establish a library of modelling patterns or templates for (re)use in a particular domain or domains, which, in many cases, would obviate the need to extend the language, using mechanisms such as stereotypes.

Finally, there is a use of stereotypes for which full-blown metamodelling is too heavyweight, and for which the template mechanism is inappropriate. This is when stereotypes are treated as pure syntactic annotations, which have no meaning for the UML modelling tool, but might have significant meaning for other tools that process the output (XMI) from the UML modelling tool. Indeed, the support that most existing modelling tools provide for UML stereotypes is of this form. If this is deemed important, then the facility can be provided simply by allowing every modelling element that can be sterotyped to have an optional attribute of type string called "stereotype".

If tagged values are also required, then the more sophisticated model can be provided by applying the template in Figure 4-1 as appropriate.




```
--stereotypes can only be applied to designated elements
context <Context> inv:
  stereoTypeable<Element>-
  >includesAll(<element>Stereotype.application.element->asSet())

--there is only one value per field in a stereotype application
--and values are only of fields defined for the stereotype
context StereotypeApplication inv:
  value.field->asSet()->asBag() = value.field and
  value.field = stereotype.field and

--field names are unique
context Stereotype inv:
  field.name->asSet()->asBag() = field.name
```

Figure 4-1 *Stereotype Template*

This template defines a stereotype to be something with a name that can be associated with named tagged fields. The application of a stereotype supplies values for the tagged fields. Tagged fields can only have strings as values, though this could be relaxed if necessary.

An application of this template would be to substitute *Context* by *Package* and *Element* by *Class*, and then *Context* by *Package* and *Element* by *Association*. Merging the results would mean that a package could define separate stereotypes to be applied to classes and associations respectively, where definitions would need to be provided for queries *stereotypeableClass* and *stereotypableAssociation*. For example, *stereotypeableClass* could be defined to return all those classes in the package or any packages nested in the package (a package defines stereotypes that can be used on any classes within its scope).

DEFINITIONS

Chapter 5
Reading Guide

The definitions part comprises a series of chapters describing the language units, which is followed by definitions of languages and definitions of templates. This is interspersed with chapters giving informal introductions to the languages and language units being defined. Each language unit/language/template is described in a separate chapter which has the following format:

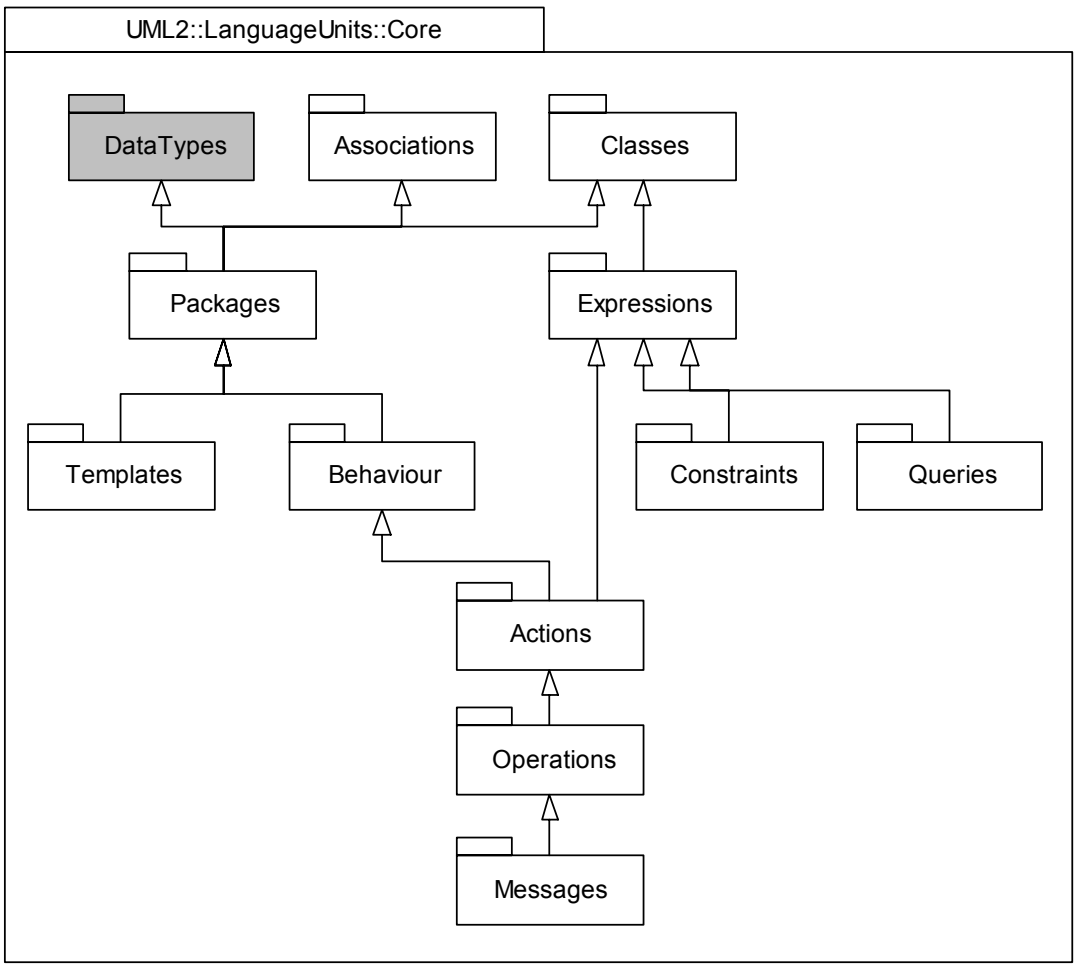
- Position in architecture
- Abstract syntax
  - Deivation from templates
  - Expansion of metamodel itself
    - class diagram
    - well-formedness rules in OCL
    - query operations defined in OCL
- Semantic domain
  - Deivation from templates
  - Expansion of metamodel itself
    - class diagram
    - well-formedness rules in OCL
    - query operations defined in OCL
- Semantic mapping (between abstract syntax and semantic domain)
  - Deivation from templates
  - Expansion of metamodel itself
    - class diagram
    - well-formedness rules in OCL
    - query operations defined in OCL

In diagrams, we have generally omitted to declare the full path names of packages – to do so is cumbersome. The "position in Architecture" section clarfies the location of packages representing language units and languages. Templates are either from *Foundation::Templates* or from *UML::Templates*.

Chapter 6
DataTypes

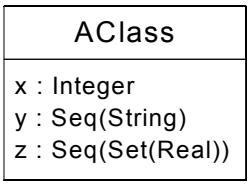
The DataTypes package defines the primitive data types supported by UML (such Integers and Strings) and collection types (Sets, Sequences and Bags).

6.1 POSITION IN ARCHITECTURE

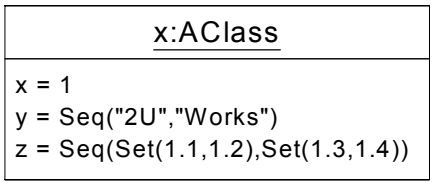


6.1.1 Example

Data types are typically used for declaring the types of attributes. For example, the following diagram shows a class with three attributes of type Boolean, Seq (String) and Seq(Set(Real)):



Values of types are described by the basic values and collection values defined in the semantic domain package. For example, values of the type Integer are the set of all integer values (1,2,3,..), whilst values of the type Seq(T) are the set of all ordered values of type T. An object of the class AClass (shown above) might have the following values for its attributes:



6.2 ABSTRACT SYNTAX

6.2.1 Derivation

This package is derived from no other packages or templates.

6.2.2 Model

The model in Figure 6-1 on page 53 shows the datatypes that can occur in a UML model. The basic type is the UML Classifier, which includes all subtypes of Classifier from the UML infrastructure.

BagType

A bag type is an unordered collection type which describes a multiset of elements where each element may occur multiple times in the bag. Part of a bag type is the declaration of the type of its elements.

CollectionType

A collection type describes a list of elements of a particular given type. Collection types are Set, Sequence and Bag types. Part of every collection type is the declaration of the type of its elements, i.e. a collection type is *parameterized* with an element type. Note that there is no restriction on the element type of a collection type. This means in particular that a collection type may be parameterized with other collection types allowing nested collections.

Associations

elementType The type of the elements in a collection. All elements in a collection must conform to this type.

EnumerationLiteral

An enumeration literal.

Associations

elementType The type of the enumeration literal.

EnumerationType

An enumeration type describes a collection of enumeration literals, each of which may be of a different type.

Associations

enumerationLiteral The set of enumeration literals belonging to the enumeration type.

Primitive

A primitive is a basic data type, such as a boolean, string, integer or real. A primitive has a name, which is its type, e.g. (“Integer”).

SeqType

A seq type is an ordered collection type which describes a list of elements where each element may occur multiple times in the sequence. Part of a seq type is the declaration of the type of its elements.

SetType

A set type is an unordered collection type which describes a set of elements where each distinct element occurs only once in the set. Part of a set type is the declaration of the type of its elements.

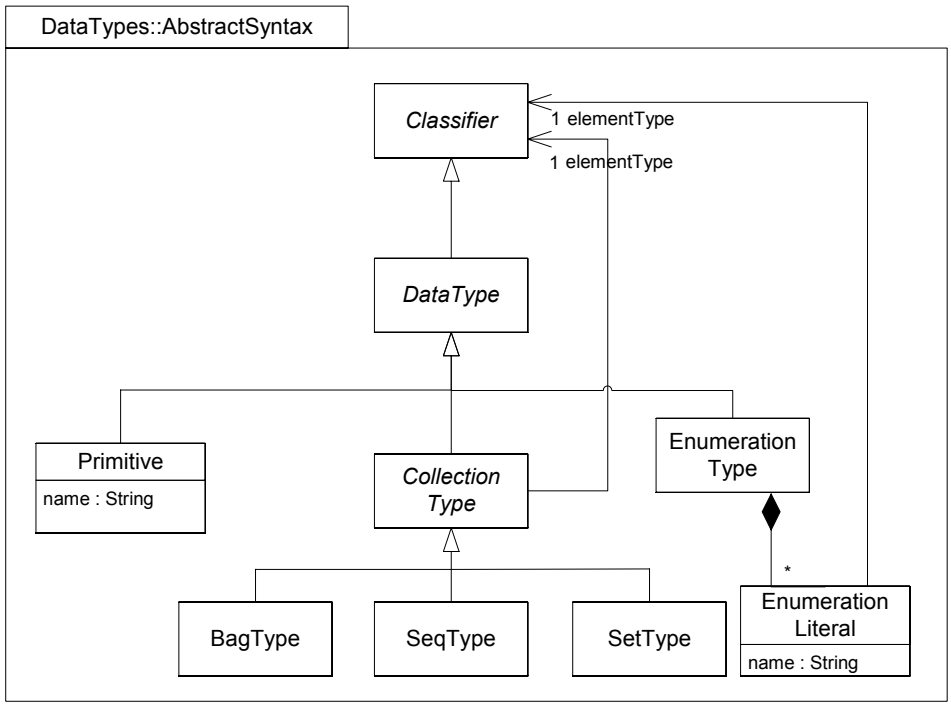


Figure 6-1 *Abstract syntax for the DataTypes package*

6.2.3 Type Conformance

The rules for checking the conformance of types are given below. Each type must define a method, *conformsTo(t)*, which returns true if the type conforms to another type, *t*.

BagType

[1] A bag type conforms to a classifier if the classifier is a bag type and their element types conform.

```
context BagType
  conformsTo(c : Classifier) : Boolean
    if c.isKindOf(BagType) then
      self.elementType.conformsTo(c.elementType)
    else
      false
    endif
```

EnumerationType

[1] An enumeration type conforms to a classifier if the classifier is an enumeration type and each of its enumeration literals conforms to a corresponding enumeration literal belonging to the classifier.

```
context EnumerationType
  conformsTo(c : Classifier) : Boolean
    if c.isKindOf(EnumerationType) then
      c.enumerationLiteral->forAll(e |
        self.enumerationLiteral->exists(e' |
          e'.elementType.conformsTo(e.elementType)
    else
      false
    endif
```

Primitive

[1] A primitive conforms to a classifier if the classifier is a primitive and has the same name. An integer may also conform to a real.

```
context Primitive
  conformsTo(c : Classifier) : Boolean
    if c.isKindOf(Primitive) then
      self.name = c.name or self.name = "Integer" and c.name = "Real"
    else
      false
    endif
```

SeqType

[1] A seq type conforms to a classifier if the classifier is a seq type and their element types conform.

```
context SeqType
  conformsTo(c : Classifier) : Boolean
    if c.isKindOf(SeqType) then
      self.elementType.conformsTo(c.elementType)
    else false
    endif
```

SetType

[1] A set type conforms to a classifier if the classifier is a set type and their element types conform.

```
context SetType
  conformsTo(c : Classifier) : Boolean
```

```
if c.isKindOf(SetType) then
  self.elementType.conformsTo(c.elementType)
else false
endif
```

6.3 SEMANTIC DOMAIN

The model in Figure 6-2 on page 56 describes the values that form the semantic domain of the UML types package. The basic type is the class Value, which includes all values of the elements described in the abstract syntax package. There is a special sub-class of the class Value called UndefinedValue, which is used to represent the undefined value for any type in the abstract syntax.

6.3.1 Derivation

This package is not derived from any other packages or templates.

6.3.2 Model

BagTypeValue

A bag type value is a collection value. It contains a set of elements, where more than one element may have the same value. Bag type values are unordered.

CollectionTypeValue

A collection type value contains a collection of elements.

Associations

elements The elements in a collection.

Element

An element representing a component of a collection. An element has a value. An element identifies the position of a element in a sequence by its indexNo. It also provides a count of the number of identical elements in a bag.

EnumerationTypeValue

An enumeration type value is a collection of enumeration literal values.

Associations

enumerationLiteralValue The set of enumeration literal values.

EnumerationLiteralValue

An enumeration literal value.

Associations

value The value of the enumeration literal value.

SeqTypeValue

A seq type value is a collection value. It contains a set of elements, where more than one element may have the same value. Sequence type values are ordered.

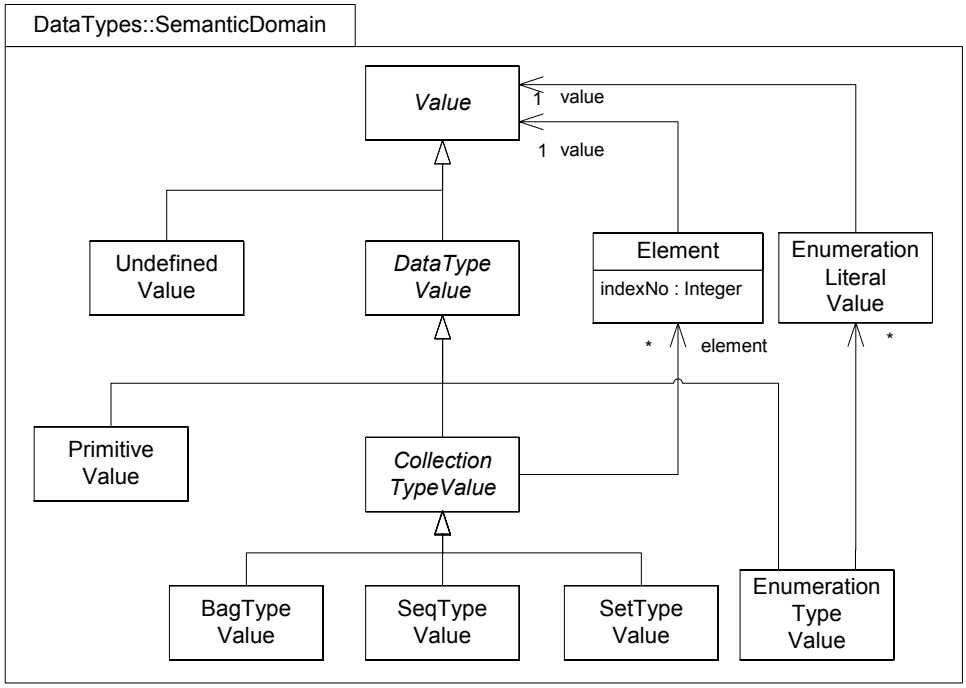


Figure 6-2 *Semantic domain for the DataTypes package*

SetTypeValue

A set type value is a collection value. It contains a set of elements, where each distinct element occurs only once in the set. Set type values are unordered.

6.3.3 Well-formedness rules

SeqTypeValue

[1] All elements belonging to a sequence have unique index numbers

```
context SeqTypeValue
  self.element -> forAll(e1, e2 | e1 <> e2 implies
    e1.indexNo <> e2.indexNo)
```

SetTypeValue

[1] All elements belonging to a set have unique values

```
context SetTypeValue
  self.element -> forAll(e1, e2 | e1 <> e2 implies
    e1.value <> e2.value)
```

6.4 SEMANTIC MAPPING

Each type has a counterpart value. A value is a valid "value" of the type if its well-formedness rules are satisfied. For example, a set type value is a valid value of a set type if its elements are valid values of the set type's element type.

6.4.1 Derivation

The semantic mapping package extends the abstract syntax and semantic domain packages of the types package with associations between semantic domain and abstract syntax elements. These associations are derived from the Semantics template as shown in 6-3.

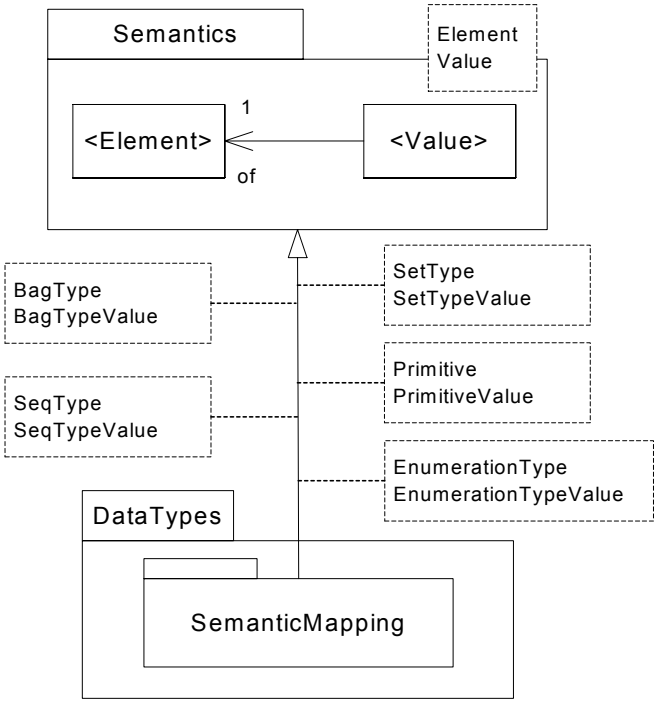


Figure 6-3 *Derivation of semantic mapping package*

6.4.2 Model

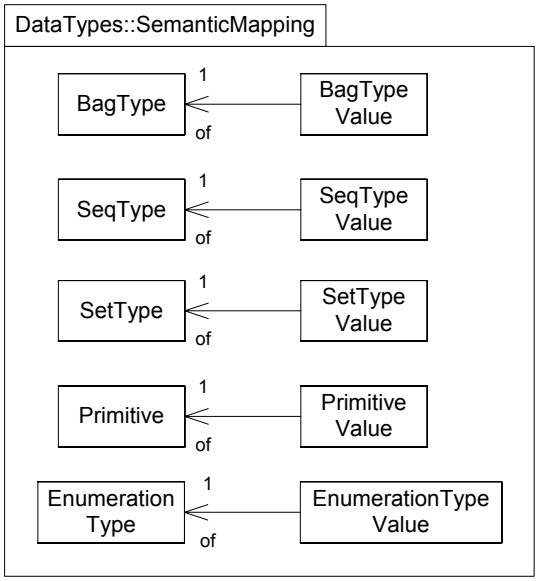


Figure 6-4 *Semantic mapping for the DataTypes package*

6.4.3 Well-formedness rules

CollectionTypeValue

[1] The elements of a collection type value must be values of the element type of the collection type.

```
context CollectionTypeValue
  self.element -> forAll(e | e.value.of = self.of.elementType)
```

EnumerationLiteralValue

[1] The value of an enumeration literal value must be a value of the element type of the enumeration literal.

```
context EnumerationLiteralValue
  self.of.elementType = self.value.of
```

EnumerationTypeValue

[1] There is an enumeration literal value for every enumeration literal belonging to the enumeration type.

```
context EnumerationTypeValue
  self.of.enumerationLiteral = self.enumerationLiteralValue.of
```

6.5 EXAMPLE SNAPSHOTS

Figure 6-5 on page 59 shows a snapshot with a set type whose element type is an Integer, and a set type value containing two primitive values that is a valid value of the set type. Note, as shown here, each element value must be unique.

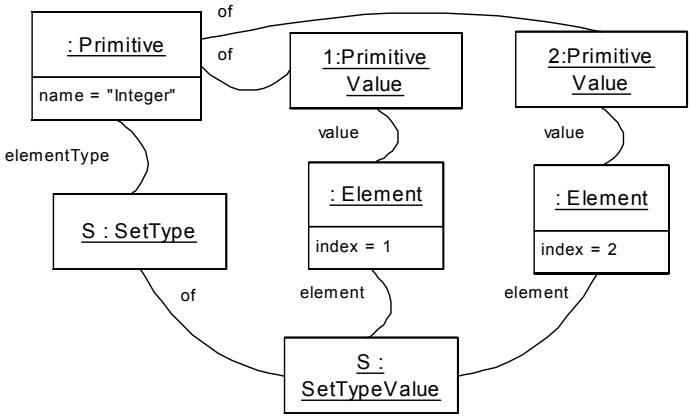


Figure 6-5 *Snapshot of a set type*

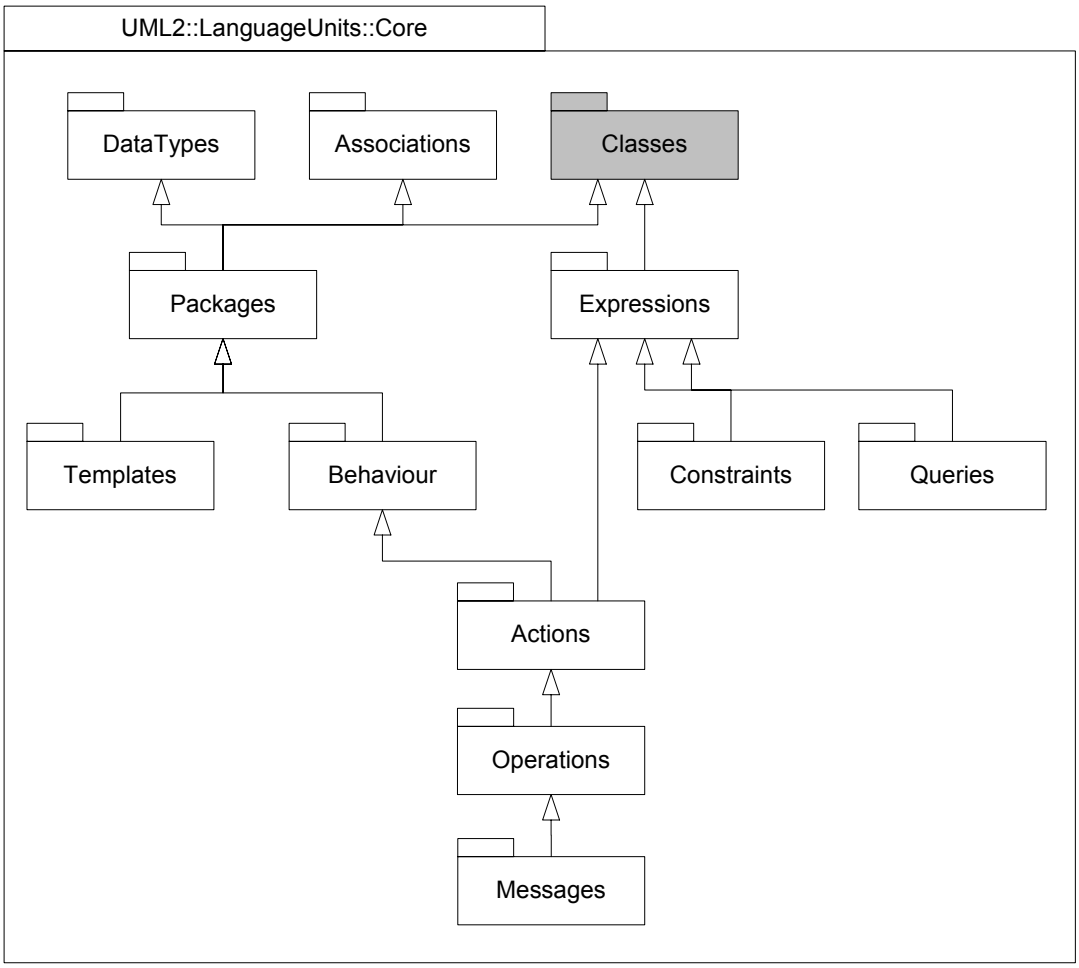
6.6 CHANGES FROM UML 1.4

The class Instance has been renamed to Value as the term "instance" was found to be generally confusing. Collection types have been added to provide support for OCL collections. The abstract association between the classes Instance and Classifier has been replaced by a uni-directional "of" association from elements in the semantic domain to elements in the abstract syntax.

Chapter 7
Classes

This package defines the abstract syntax and semantics of the static features of classes (operations and queries are dealt with in later chapters). Classes describe the possible states of the system in terms of objects. An object is a value or instance of a class. The structure of each class is described in terms of a set of attributes. An attribute has a type, which specifies the values that can be assigned to its class's objects. Classes also support the notion of generalization: the ability to reuse structural definitions from one class (the parent, or super-class) in another (the child, or sub-class).

7.1 POSITION IN ARCHITECTURE



7.1.1 Example

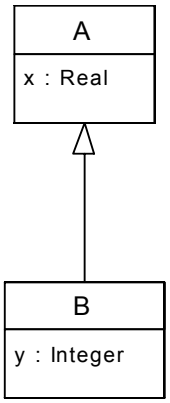


Figure 7-1 *Classes example*

An example of a pair of classes is shown in figure 7-1 on page 61. In this model, class A has an attribute *x* which is of type Real while class B has an attribute y which is of type Integer. Class B specializes class A.

7.2 ABSTRACT SYNTAX

7.2.1 Derivation

Figure 7-2 on page 62 shows the derivation of the Classes abstract syntax package using the structural feature-classifier and multiplicity templates. A class is a namespace for its structural features (its members). The members of a class's namespace include its owned and inherited structural features. Classes are generalizable. A generalisation relationship results in all members of the parent namespace being inherited by the child. Attributes are structural features and have a name and a type. Attributes have an optional multiplicity. A multiplicity is a set of integer values including the distinguished value "unLimited" and defines the range of values that can be assigned to an attribute.

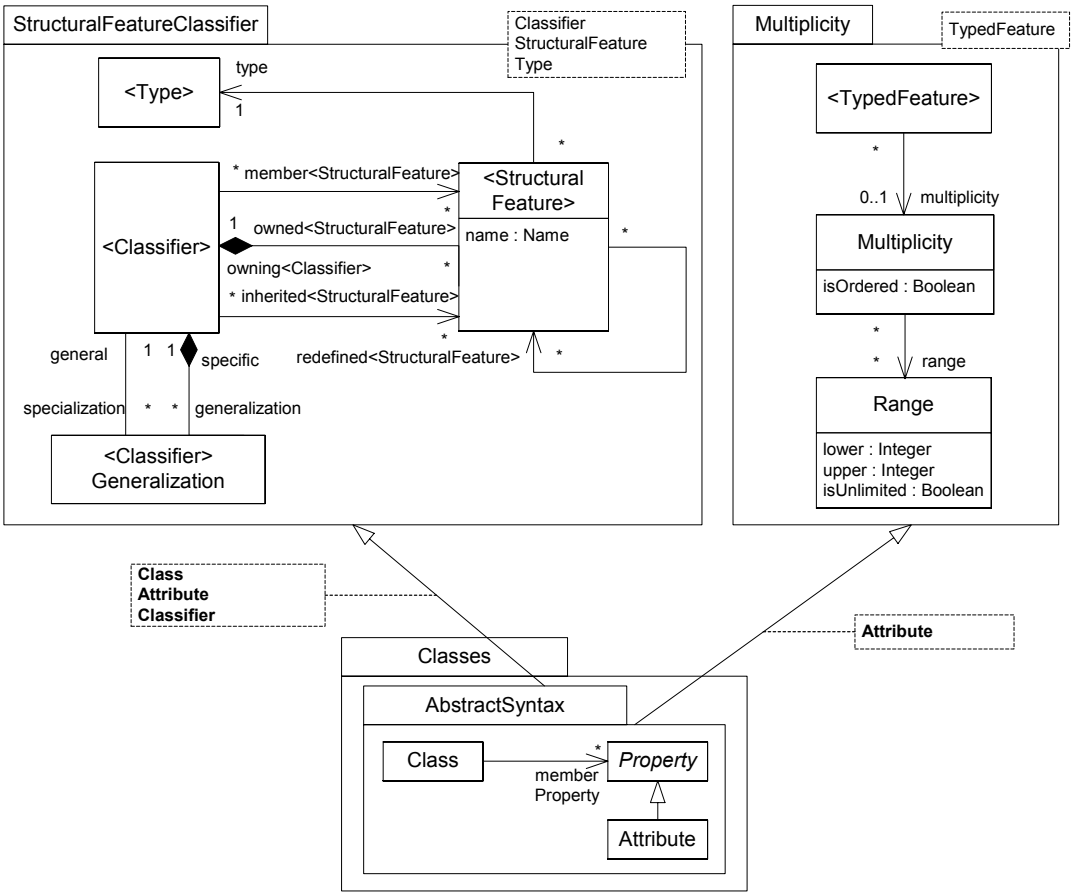


Figure 7-2 *Derivation of Classes abstract syntax package*

7.2.2 Model

Figure 7-3 on page 63 shows the abstract syntax of the classes package. A class is a namespace for its attributes (its members). The members of a class's namespace include its owned and inherited attributes. Classes may specialize other classes, in which case, all members of the parent classes namespace are inherited by the child classes. Attributes have a name and a type. Attributes can also be redefined in a generalization relationship. Redefinition allows the name of an attribute to be changed by the redefining attribute but the types of the atributes must conform. Attributes also have an optional multiplicity. A multiplicity is a set of integer values including the distinguished value "unLimited" that specifies whether an attribute is multi-valued and what the range of its values can be.

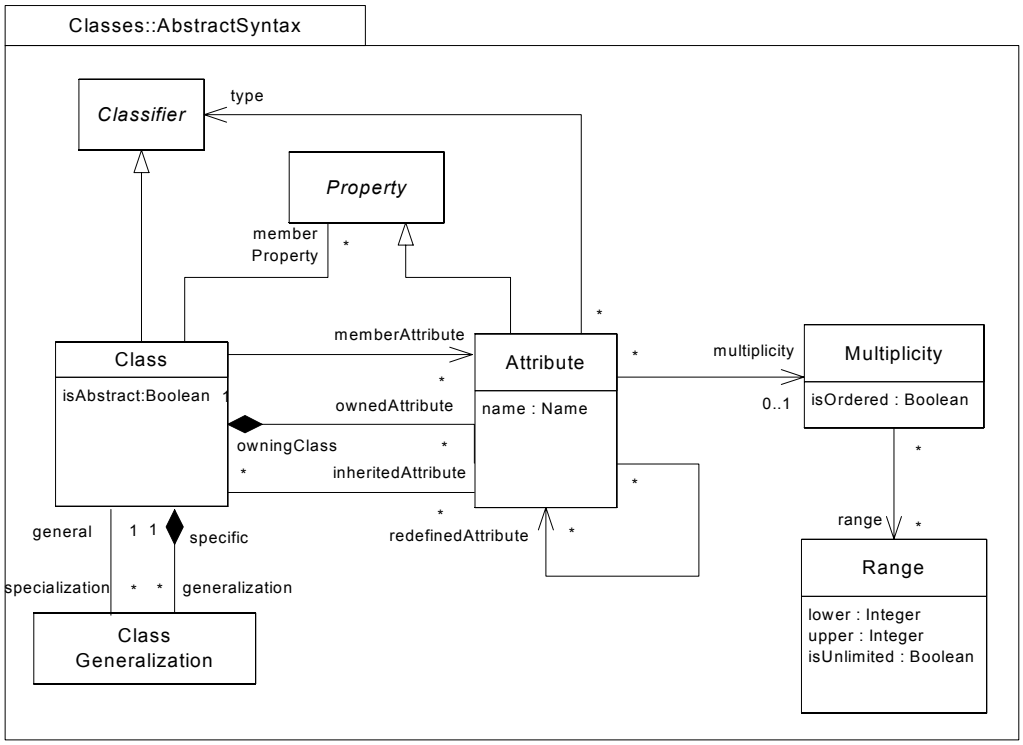


Figure 7-3 *Abstract syntax for Classes package*

Attribute

Attributes define the type of the values that can be stored in the objects of a class. Attributes have a name, a type, and an optional multiplicity. If an attribute has a multiplicity, the attribute's type is defined to be a set (or a sequence if the multiplicity is ordered). Attributes may redefine their parent classes' attributes. A redefined attribute may have a different name to the attribute it redefines, but their types must be conformant. An attribute is a property, which means that they can be referenced through a property call expression (see Expressions chapter).

Attributes

name The name of the attribute.

Associations

multiplicity Specifies the range of values of the attribute.

owningClass The class that owns the Attribute.

redefinedAttribute The attribute that the attribute redefines.

type The type of the attribute.

Class

A class describes the structure of its values in terms of attributes. Classes permit the reuse of their parent classes' features through specialization. A class inherits its parents member attributes into its namespace provided that they have not been redefined.

Associations

generalization All generalization relationships that generalize the class. The generalization relationship navigates to the class that is the more general (parent) class.

ownedAttribute The attributes owned by the class.

memberAttribute The attributes that can be viewed as being in its namespace of the class, including its owned, inherited and imported attributes.

memberProperty The properties that can be viewes as being in its namespace of the class - this must include all member attributes as well as queries (see Chapter 14)

inheritedAttribute The attributes inherited from the class's parents.

isAbstract True if the class is abstract

specialization All specialization relationships that specialize the class. The specialization relationship navigates to the class that is the more specific (child) class.

ClassGeneralization

A generalization relationship between classes.

Associations

general The class that is the more general (parent) class in the relationship.

specialization The class that is the more specific (child) class in the relationship.

Multiplicity

Specifies the number of elements that may be assigned to a value of an attribute.

Attributes

isOrdered True if the elements are to be ordered.

Associations

range The set of number ranges belonging to the multiplicity.

7.2.3 Well-formedness Rules

Attribute

[1] An attribute's type must conform to the type of its redefined attributes.

```
context Attribute inv:
  self.redefinedAttribute->forAll(f |
    self.type.conformsTo(f.type))
```

[2] If an attribute has a multiplicity, its type must be of the appropriate collection type.

```
context Attribute inv:
  if self.multiplicity <> null then
    if self.multiplicity.isOrdered then
      self.type.isKindOf(Core::DataTypes::SetType)
        else
          self.type.isKindOf(Core::DataTypes::SeqType)
    endif
  endif
```

Class

[1] Circular inheritance is not permitted.

```
context Class inv:
  not self.allGeneralElements()->includes(self)
```

[2] The member attributes of a class include its owned and inherited attributes.

```
context Class inv:
  self.memberAttribute->includesAll(self.ownedAttribute ->
    union(self.inheritedAttribute))
```

[3] Attributes cannot be owned and inherited.

```
context Class inv:
  self.ownedAttribute->intersection(self.inheritedAttribute) -> isEmpty
```

[4] A class cannot have two attributes with the same name.

```
context Class inv:
  self.memberAttribute->forAll(e1|
    self.memberAttribute->forAll(e2|
      e1 <> e2 implies e1.name <> e2.name))
```

[5] The inherited members of a class are the attributes of its parents classes that aren't redefined.

```
context Class inv:
  self.inheritedAttribute = self.generalElements()->iterate(p s = Set{} |
     s->union(p.memberAttribute->reject(c |
       self.memberAttribute -> exists(c' |
         c'.redefinedAttributes->includes(c)))))
```

[6] A class's attributes may only redefine its parent classes attributes.

```
context Class inv:
  self.memberAttribute -> forAll(a |
    self.generalElements()-> collect(g | g.memberAttributes) ->
      includesAll(a.redefinedAttribute))
```

[7] The member properties of a class include all its member attributes.

```
context Class inv:
  self.memberProperty->includesAll(self.memberAttribute)
```

7.2.4 Operations

Class

[1] A class conforms to another class if it specializes the class or is the same class.

```
context Class::conformsTo(c : Class):Boolean
  self.generalElements()->includes(c) or self = c
```

[2] Returns the parents of a class.

```
context Class::generalElements():Set(Class)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[3] Transitively returns all parents of a class.

```
context Class::allGeneralElements():Set(Class)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

[4] Looks up an attribute in a class when given a name.

```
context Class::lookupAttributeforName(x : Name):Attribute
```

```
self.memberAttribute->select(e| e.name = x ).selectElement()
```

[5] Looks up an attribute's name when given the attribute.

```
context Class::lookupNameForAttribute(x : Attribute):Name
  self.memberAttribute->select(e|e = x ).selectElement().name
```

7.3 SEMANTIC DOMAIN

7.3.1 Derivation

Figure 7-4 on page 66 shows the derivation of the Classes semantic domain package from the structural feature classifier value template. A classifier value is a value of a classifier and contains a set of static structural feature values.

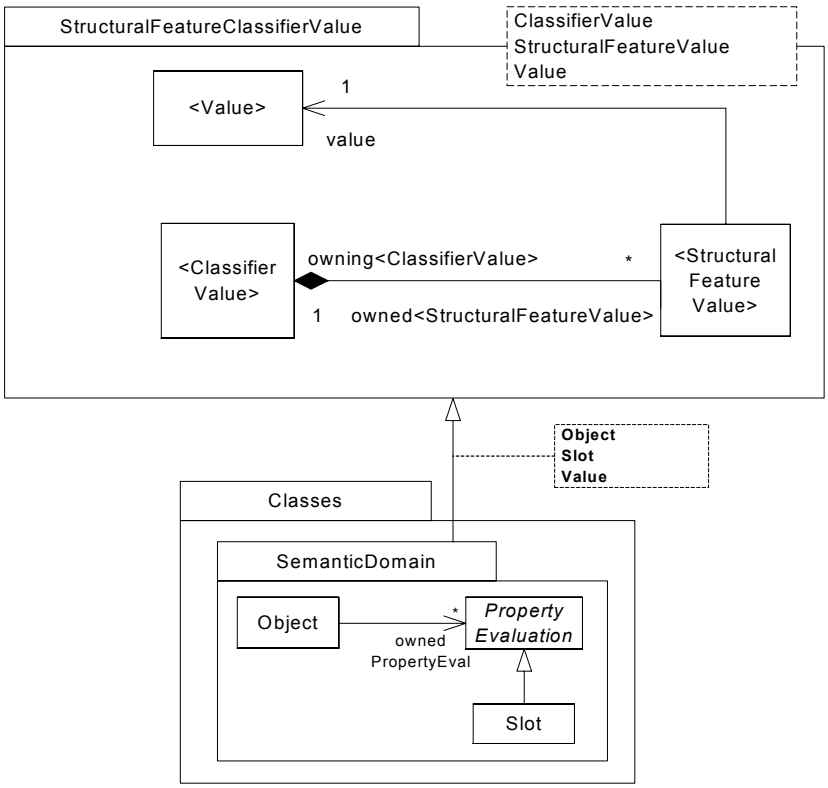


Figure 7-4 *Derivation of Classes semantic domain package*

7.3.2 Model

The semantic domain of the classes package is shown in 7-5 on page 67. It defines the fundamental concepts that are necessary to express the static meaning of classes. An object is a value or instance of a class. The state of an

object is described by its slots. A slot is a value of an attribute. It contains a reference to a value, which is the value that is assigned to the slot.

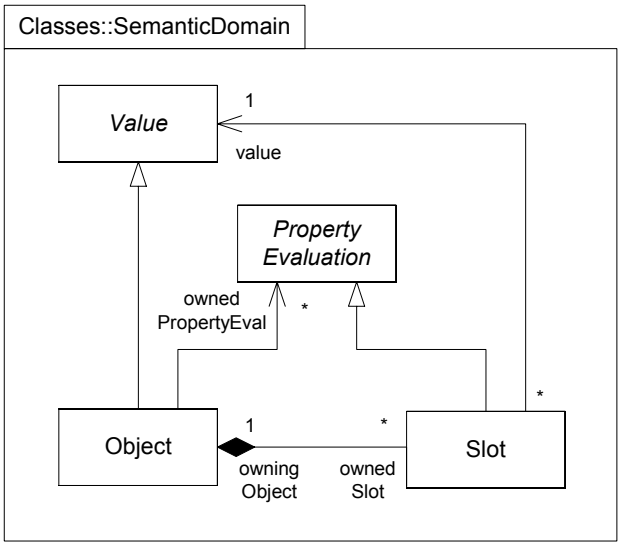


Figure 7-5 *Semantic domain for the Classes package*

Object

Objects are containers of slots.

Associations

ownedSlot The slots owned by the object.

ownedPropertyEval The property evaluations (including slots) that are owned by the object.

Slot

Slots represent the data values of an object. A slot is a property evaluation, which means that it can be accessed through a property call evaluation (see Expressions chapter).

Associations

value The value of the slot.

7.3.3 Well-formedness Rules

Object

[1] The owned property evaluations of an object includes all its slots..

```
context Object inv:
  self.ownedPropertyEval->includesAll(self.ownedSlot)
```

7.4 SEMANTIC MAPPING

7.4.1 Derivation

The structural feature semantics template is used to derive the semantic mapping for the classes package as shown in figure 7-6 on page 68. This template ensures that each element in the semantic domain is mapped to their appropriate abstract syntax element and that the necessary constraints on their relationships are also generated.

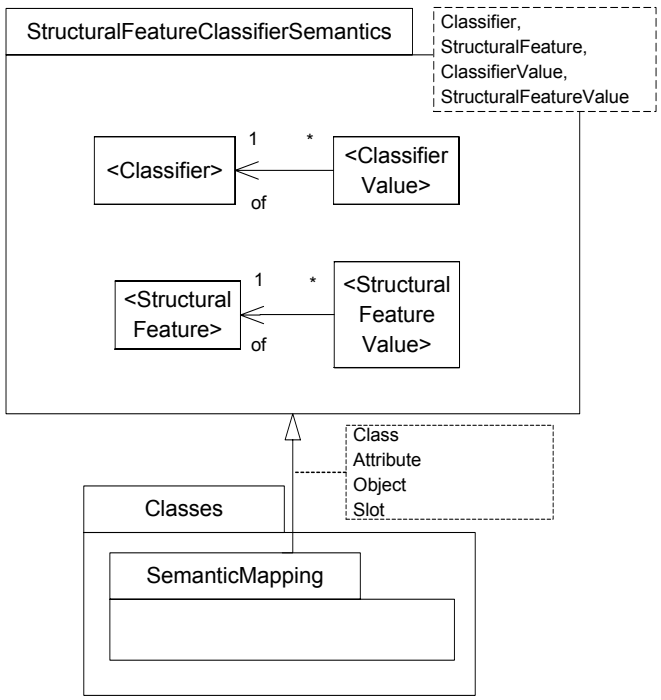


Figure 7-6 *Derivation of the Classes semantic mapping package*

7.4.2 Model

The semantics mapping package of the classes package is shown in Figure 7-7 on page 68. It defines the relationship that holds between classes and attributes and their values: objects and slots. An object is a value of a class. The meaning of a class is defined by the set of objects that are its valid values. The state of an object is described by its slots. A slot is a value of an attribute. For an object to be a valid value of a class then it must contain slots for each of the attributes in the namespace of the class and vice versa. Furthermore, the value of a slot must be a value of the type of its attribute.

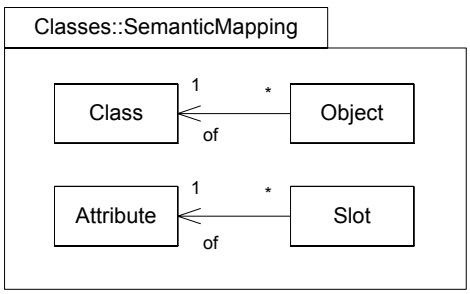


Figure 7-7 *Semantic mapping for the Classes package*

7.4.3 Well-formedness rules

Object

[1] An object should contain a slot for all attributes in the object's class's namespace.

```
context Object inv:
  self.of.memberAttribute->forAll(c |
    self.ownedSlot->exists(d | d.of = c))
```

[2] For each slot owned by an object there should be an attribute of the object's class's namespace that the slot is a value of.

```
context Object inv:
  self.ownedSlot->forAll(c |
    self.of.memberAttribute->exists(d | c.of = d))
```

[3] For each property evaluation owned by an object there should be a property of the object's class's namespace that the property evaluation is a value of.

```
context Object inv:
  self.ownedPropertyEvaluation->forAll(pv |
    self.of.memberProperty->exists(p | pv.of = p))
```

[4] Objects cannot be instances (values) of abstract classes.

```
context Object inv:
  not self.of.isAbstract
```

Slot

[1] The value of a slot should be a value of the type that conforms to the slot's attribute.

```
context Slot inv:
  self.value.of.conformsTo(self.of.type)
```

[2] The values of a slot should match the multiplicity of the slot's attribute.

```
context Slot inv:
  if self.of.multiplicity <> null then
    self.of.multiplicity.range->exists(mr |
      self.value.element->collect(e | e.value)->size >= mr.lower and
      (mr.isUnlimited or
      (not mr.isUnlimited and
       self.value.element->collect(e | e.value)->size <= mr.upper)))
  else
    true
  endif
```

7.5 EXAMPLE SNAPSHOTS

The model in figure 7-8 on page 70 is instantiated and the resulting snapshot shown in 7-9 on page 70. Class B is a specialization of class A and therefore attribute x is inherited into the namespace of class B.

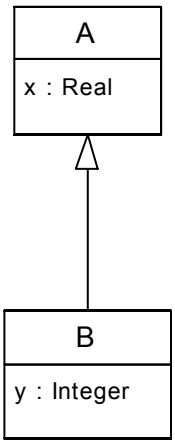


Figure 7-8 *Example classes*

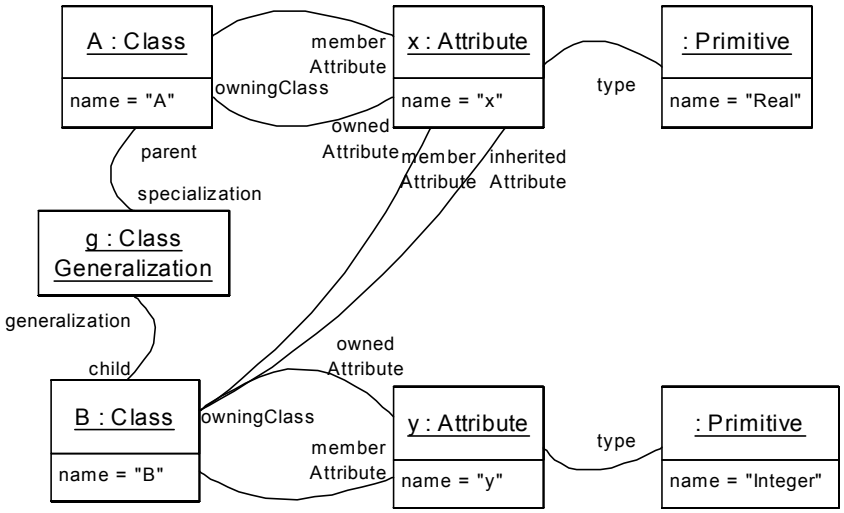


Figure 7-9 *Snapshot of Figure 7-8 on page 70*

Figure 7-10 on page 71 shows what happens when the attribute y redefines the attribute x. In this case, x is no longer required to be inherited by the class B. The redefinition is permitted becuase the type of attribute y (Integer) conforms to the type of attribute x (Real).

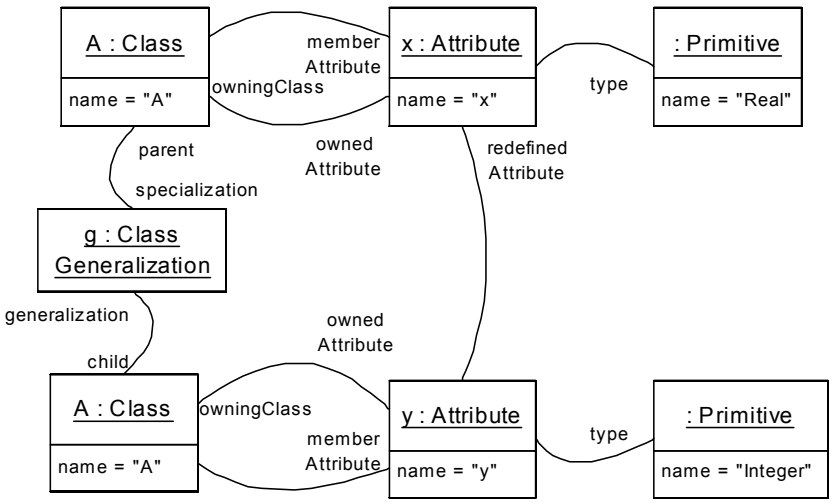


Figure 7-10 *Snapshot of y redefines x*

Figure 7-11 on page 71 shows an object that is a valid instance of class B from figure 7-9 on page 70. It has a two slots, one for attribute x which has the instance of a real type as its value, and one for the slot of the inherited Attribute y. It is important to note that the inheritence has been flattened out and the slots corresponding to inherited attributes also become owned slots of the object.

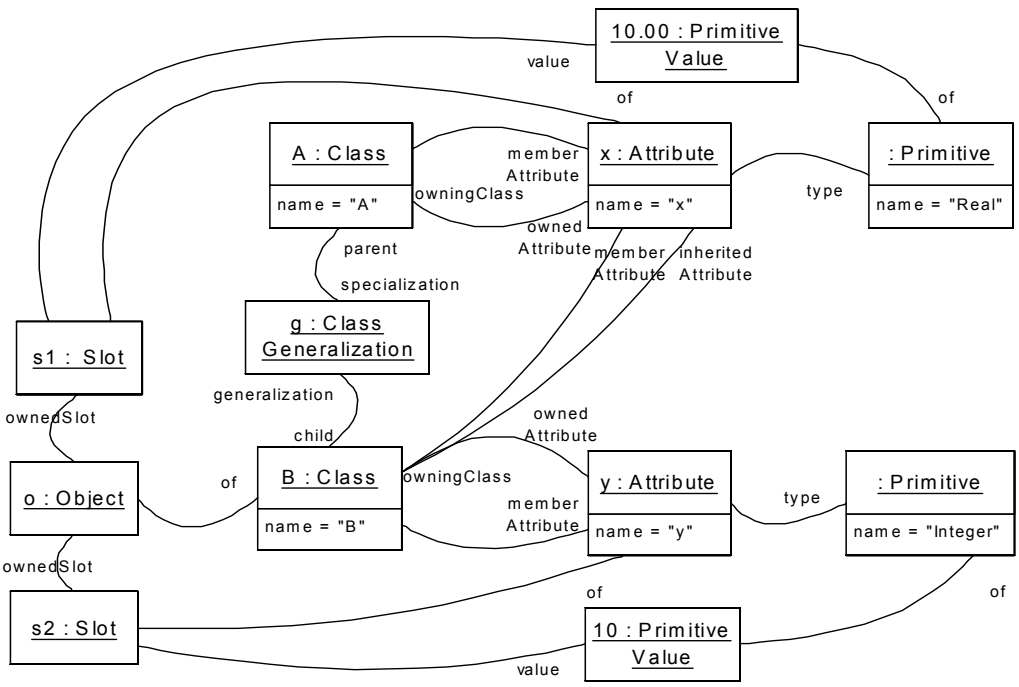


Figure 7-11 *Snapshot with Object of Class B*

7.6 CHANGES FROM UML 1.4

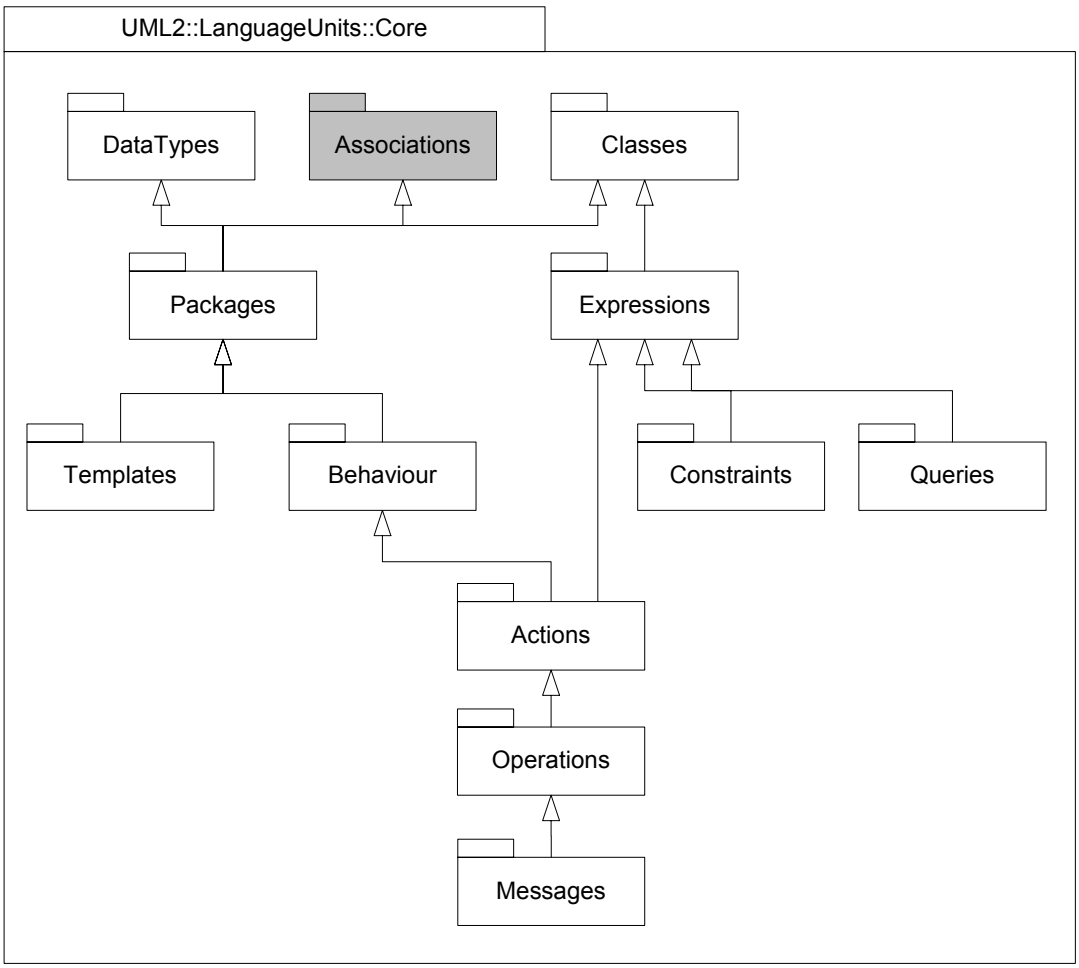
Redefinable features are not a part of UML 1.4.
AttributeLink has been replaced by Slot.

Chapter 8
Associations

This package defines the abstract syntax and semantics of associations. Associations describe static relationships between classes. The meaning of an association is defined in terms of links between objects. Associations have association ends that specify the types of objects that they link and the number of links that can exist between specific objects. Associations are also generalizable: thus permitting the reuse of the features of one association (the parent, or super-class) in another (the child, or sub-class).

In this chapter, an alternative (and equivalent) semantics for associations is described via a translation from navigable association ends to pairs of attributes or queries.

8.1 POSITION IN ARCHITECTURE



8.1.1 Example

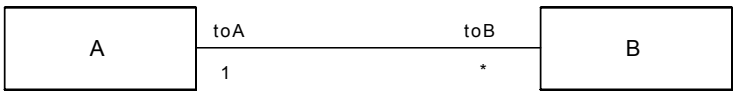


Figure 8-1 *An example of an assocation between two classes*

Figure 8-1 on page 73 shows an example of an association. It describes two classes A and B with a bidirectional navigable association between them. This association has a one to many multiplicity .

8.2 ABSTRACT SYNTAX

8.2.1 Derivation

Figure 8-2 on page 74 shows how the associations abstract syntax package is derived from the StructuralFeature-Classifier and Multiplicity templates. An association is a classifier. It is a namespace for its structural features and is generalisable. An association end is a structural feature and supports redefinition. An association end may have an (optional) multiplicity.

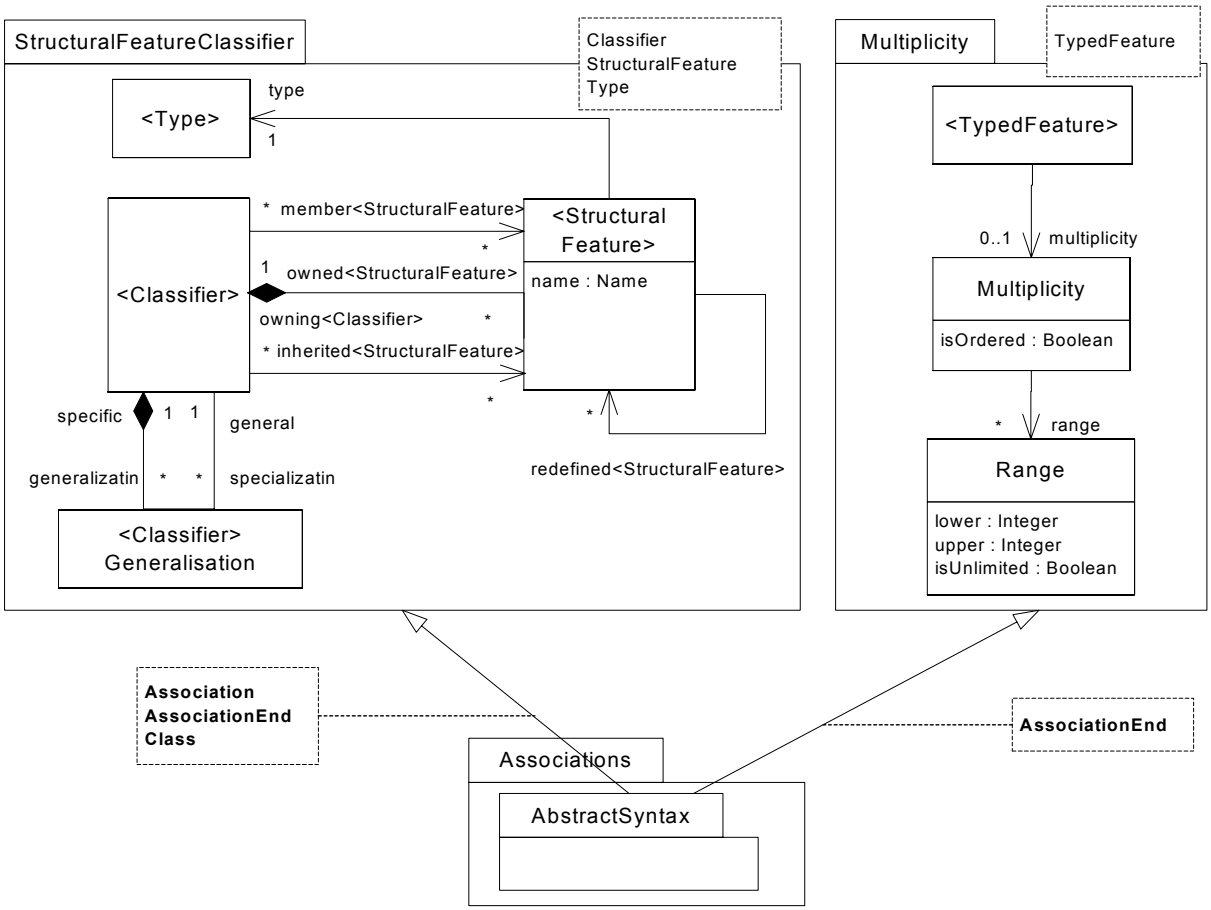


Figure 8-2 *Derivation of Associations abstract syntax package*

8.2.2 Model

Figure 8-3 on page 75 shows the abstract syntax of the associations package. An association is a namespace for its association ends. An association may have two or more association ends. An association end has a name, a type, which is the class it is connected to, and a multiplicity, which specifies how many objects an object of the class at the other end of the association end can be linked to.

Navigable ends are specializations of association ends. An equivalence mapping is defined from navigable association ends to properties. A property is the abstract superclass of an attribute and a query. This enables a navigable association end to be viewed as either an attribute or query of a class at the opposite end of the association - a common interpretation used by many modellers.

Member association ends are those association ends that belong to the association's namespace and include its owned association ends and its inherited association ends. An association has a set of generalizations that relate it to its parent associations, and set of specializations that relate it to its child associations.

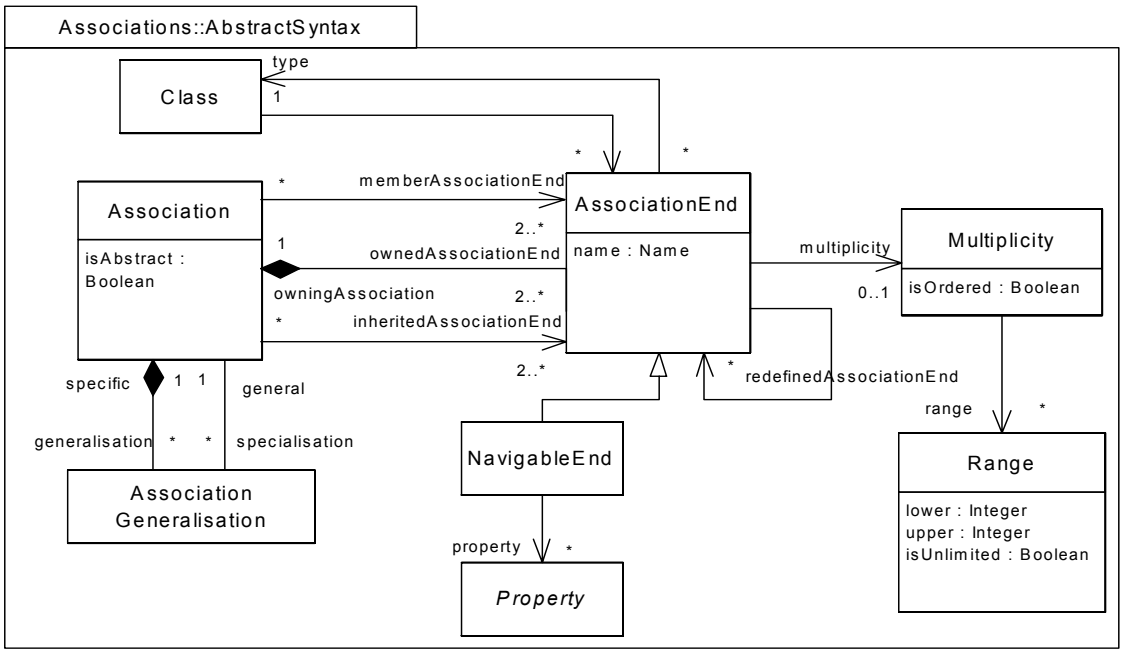


Figure 8-3 *Abstract syntax for Associations package*

Association

An association connects two or more classes and specifies a relationship between objects of these classes. Associations permit the reuse of their parent associations features through specialization. An association inherits its parents member association ends into its namespace provided that they are not redefined.

Associations

generalization All generalization relationships that generalize the association. The generalization relationship navigates to the association that is the more general (parent) association.

ownedAssociationEnd The association ends owned by the association.

memberAssociationEnd The association ends that can be viewed as being in its namespace of the association, including its owned, inherited and imported association ends.

inheritedAssociationEnd The association ends inherited from the association's parents.

isAbstract True if the association is abstract.

specialization All specialization relationships that specialize the association. The specialization relationship navigates to the association that is the more specific (child) association.

AssociationGeneralization

A generalization relationship between associations. When a association specializes another association, its parents association ends are inherited into the child's namespace.

Associations

general The association that is the more general (parent) association in the relationship.

specialization The association that is the more specific (child) association in the relationship.

AssociationEnd

An association end connects an association to a class. Its multiplicity defines the number of objects at the other ends of the association that an object of the class can be linked to. An association end can be redefined, in which case the redefining association end may have a different name to the redefined association end. However, their types must be conformant.

Attributes

name The name of the association end.

Associations

multiplicity The number of objects of the classes at the other ends of the association that an object of its class can be linked to.

redefinedAssociationEnd The association ends that the association end redefines.

type The type of the association end, i.e. the class which the association end connects to.

Multiplicity

Specifies the number of objects that an object of a class at the other at the other end of the association can be linked to.

Attributes

isOrdered True if the objects are to be ordered.

Associations

range The set of number ranges belonging to the multiplicity.

NavigableEnd

An association end that is navigable from any of the classes at the others ends of the association. A navigable end is associated with properties (attributes or a queries) that belong to the classes at the other ends of the association. Each property has the same name, multiplicity and element type as the navigable end. Classes at the other ends of the association can thus navigate to objects of the navigable end's type through these properties.

Associations

property The attributes or queries that enable classes at the other end of the association to navigate to objects of the navigable end's type.

Property

An abstract superclass for attributes and queries.

8.2.3 Well-formedness Rules

Association

[1] Circular inheritance is not permitted.

```
context Association inv:
  not self.allGeneralElements()->includes(self)
```

[2] The members of an association include its owned and inherited association ends.

```
context Association inv:
  self.memberAssociationEnd->includesAll(self.ownedAssociationEnd ->
    union(self.inheritedAssociationEnd))
```

[3] Association ends cannot be owned and inherited.

```
context Association inv:
  self.ownedAssociationEnd->intersection(self.inheritedAssociationEnd) ->
    isEmpty
```

[4] The inherited members of an association are the association ends of its parents association ends that are not redefined.

```
context Association inv:
  self.inheritedAssociationEnd = self.generalElements()->iterate(p s = Set{} |
     s->union(p.memberAssociationEnd->reject(c |
       self.memberAssociationEnd -> exists(c' |
         c'.redefinedAssociationEnd->includes(c)))))
```

[5] An association's association ends may only redefine its parent classes association ends.

```
context Association inv:
  self.memberAssociationEnd -> forAll(a |
    self.generalElements()-> collect(g | g.memberAssociationEnd) ->
      includesAll(a.redefinedAssociationEnd))
```

AssociationEnd

[1] An association end's type must conform to the type of its redefined association ends.

```
context AssociationEnd inv:
  self.redefinedAssociationEnd->forAll(f |
    self.type.conformsTo(f.type))
```

[2] An association end's multiplicity must conform to the multiplicity of its redefined parent association ends.

```
context AssociationEnd inv:
  self.redefinedAssociationEnd->forAll(f |
    self.conformsTo(f))
```

Class

[1] A class's association ends must include a reference to the class.

```
context Class inv:
  self.associationEnd -> exists(l | l.type = self)
```

NavigableEnd

[1] A navigable end is associated with properties (attributes or queries) belonging to all classes at the other ends of the association through which values of the navigable end's type can be navigated to.

```
context NavigableEnd inv:
  self.property.owningClass =
  self.otherEnd().type
```

[2] The properties of a navigable end have the same element type, multiplicity and name as the navigable end.

```
context NavigableEnd inv:
  self.property->forAll(p |
    p.type.elementType = self.type and
    p.multiplicity = self.multiplicity and
    p.name = self.name)
```

8.2.4 Operations

AssociationEnd

[1] Returns the opposite ends of the association end.

```
context AssociationEnd::otherEnd() : Set(AssociationEnd)
  self.owningAssociation.memberAssociationEnd->reject(y | y = self)
```

Association

[1] Returns the parents of an association.

```
context Association::generalElements():Set(Association)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[2] Transitively returns all parents of an association.

```
context Association::allGeneralElements():Set(Association)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

[3] Looks up an association end in a association when given a name.

```
context Association::lookupAssociationEndforName(x : Name):AssociationEnd
  self.memberAssociationEnd->select(e| e.name = x ).selectElement()
```

[4] Looks up an association end's name when given the association.

```
context Association::lookupNameForAssociationEnd(x : AssociationEnd):Name
  self.memberAssociationEnd->select(e|e = x ).selectElement().name
```

Class

[1] Returns the associations attached to the class.

```
context Class::associations():Set(Association)
  self.associationEnd->collect(x | x.owningAssociation)
```

[2] Returns the opposite association ends attached to the class.

```
context Class::oppositeAssociationEnds():Set(AssociationEnd)
  self.associations()->iterate(x s = Set{} |
  s->union(x.memberAssociationEnd->reject(y | y.type = self))
```

Multiplicity

[1] Returns true if a multiplicity conforms to another multiplicity.

```
context Multiplicity::conformsTo(x : Multiplicity):Boolean
  TBD.
```

8.3 SEMANTIC DOMAIN

8.3.1 Derivation

Figure 8-6 on page 82 shows the derivation of the Associations semantic domain package from the structural feature classifier value template. A classifier value is a value of a classifier and contains a set of static structural feature values.

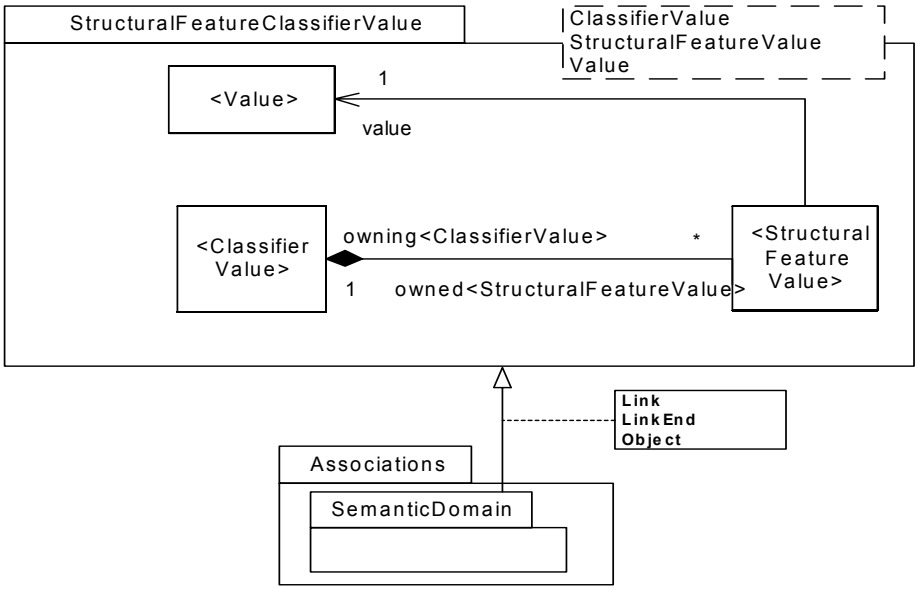


Figure 8-4 *Derivation of Classes semantic domain package*

8.3.2 Model

The semantic domain of the associations package is shown in 8-5 on page 80. A link is a value of an association. A link relates objects of the classes connected by the association. A link contains link ends. A link end is a value of an association end. A navigable link end is a link end whose value can be navigated to from a property evaluation (a slot or query evaluation) belonging to the objects at the other end of the link.

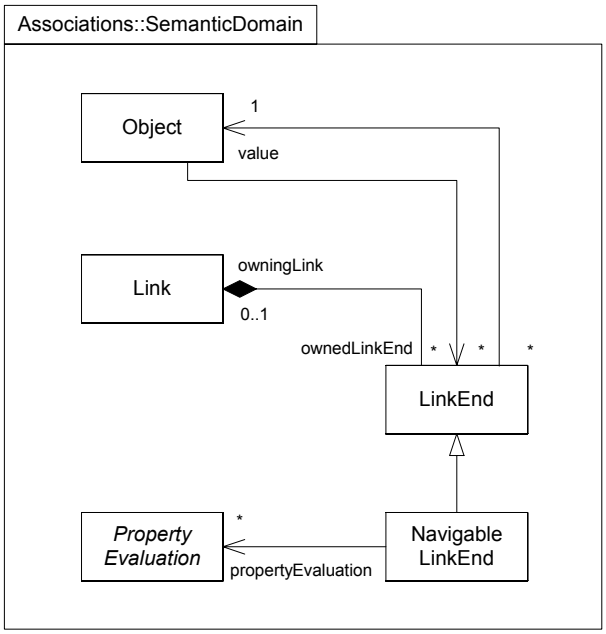


Figure 8-5 *Semantic domain for the Associations package*

Link

Links contain link ends.

Associations

ownedLinkEnd The link ends owned by the object.

LinkEnd

Link ends represent the values of an link.

Associations

value The value of the link end.

NavigableLinkEnd

Navigable link ends represent the values of a link that can be navigated to from a property evaluation (slot or query) belonging to an object at the opposite end of the link.

Associations

value The value of the navigable link end.

Object

An object.

Associations

linkEnd The linkends that the object is attached to.

8.3.3 Well-formedness Rules

NavigableLinkEnd

[1] A navigable link end is associated with property evaluations (slots or query evaluations) belonging to all objects at the other ends of the link through which the navigable link end's value can be navigated to.

```
context NavigableLinkEnd inv:
  self.propertyEvaluation.owningObject =
  self.otherEnd().value
```

[2] The property evaluations of a navigable link end include the navigable link end's value.

```
context NavigableLinkEnd inv:
  self.propertyEvaluation->forAll(p |
    p.value.element.value->includes(self.value))
```

Object

[1] An object's link ends must include a reference to the object.

```
context Object inv:
  self.linkEnd -> exists(l | l.value = self)
```

8.3.4 Operations

LinkEnd

[1] Returns the opposite ends of the link end.

```
context LinkEnd::otherEnd() : Set(LinkEnd)
  self.owningLink.ownedLinkEnd->reject(y | y = self)
```

Object

[1] Returns the links that are attached to the object.

```
 context Object::links() : Set(Link)
  self.LinkEnd -> collect(x | x.owningLink)
```

[2] Returns the opposite link ends to the object.

```
context Object::oppositeLinkEnds() : Set(LinkEnd)
  self.links()->iterate(x s = Set{} |s -> union(x.ownedLinkEnd->
    reject(y | y.value = self)))
```

8.4 SEMANTIC MAPPING

8.4.1 Derivation

The template used to stamp out the semantic mapping for the associations package is shown in figure 8-6 on page 82. This ensures that each element in the semantic domain is mapped to their appropriate abstract syntax element and that the necessary constraints on their relationships are stamped out.

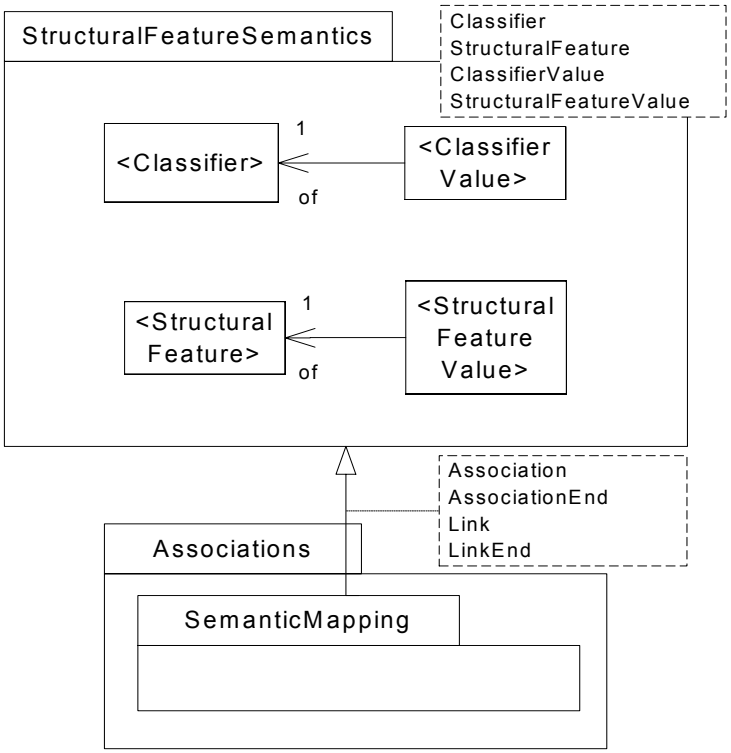


Figure 8-6 *Derivation of the Associations semantic mapping package*

8.4.2 Model

The semantics mapping package of the associations package is shown in Figure 8-7 on page 82. A link is a value of an association. A link end is a value of an association end. A link must contain link ends for each of the attributes owned by its association and vice versa. The value of a link end must be a value of the type of its association end.

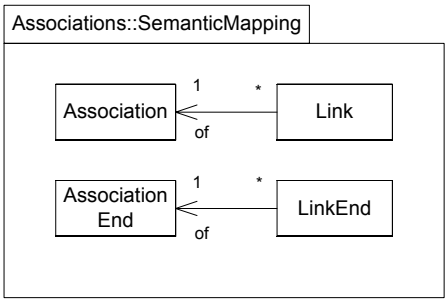


Figure 8-7 *Semantic mapping for the Associations package*

8.4.3 Well-formedness rules

Link

[1] A link should contain a link end for all association ends in the link's association's namespace.

```
context Link inv:
  self.of.memberAssociationEnd->forAll(c |
    self.ownedLinkEnd->exists(d | d.of = c))
```

[2] For each link end owned by a link there should be an association end of the link's association's namespace that the link end is a value of.

```
context Link inv:
  self.ownedLinkEnd->forAll(c |
    self.of.memberAssociationEnd->exists(d | c.of = d))
```

[3] Links cannot be values of abstract associations.

```
context Link inv:
  not self.of.isAbstract
```

LinkEnd

[1] The value of a link end should be a value of the type that conforms to the link end's association end's type.

```
context LinkEnd inv:
  self.value.of.conformsTo(self.of.type)
```

NavigableLinkEnd

[1] The property evaluations of a navigable link end must commute with its navigable end's properties.

```
context NavigableLinkEnd inv:
  self.of.property = self.propertyEvaluation.of->asSet
```

Object

[1] The number of objects at the opposite link ends of the object must conform to the opposite association ends multiplicity.

```
context Object inv:
  self.of.oppositeAssociationEnds()->forAll(ae |
    ae.multiplicity.range->exists(mr |
      self.selectedLinkEnds(ae)->size >= mr.lower and
      (mr.isUnlimited or
      (not mr.isUnlimited and
       self.selectedLinkEnds(ae)->size <= mr.upper))))
```

8.4.4 Operations

```
context Object::selectedLinkEnds(ae : AssociationEnd) : Set(LinkEnd)
  self.oppositeLinkEnds()->select(x | x.of = ae)
```

8.5 EXAMPLE SNAPSHOTS

Figure 8-9 on page 84 is a snapshot of the association of figure 8-8 on page 84. The navigable association ends of the association are associated with two attributes that the opposite ends of the association can be navigated through.

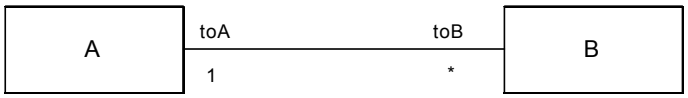


Figure 8-8 *Association example*

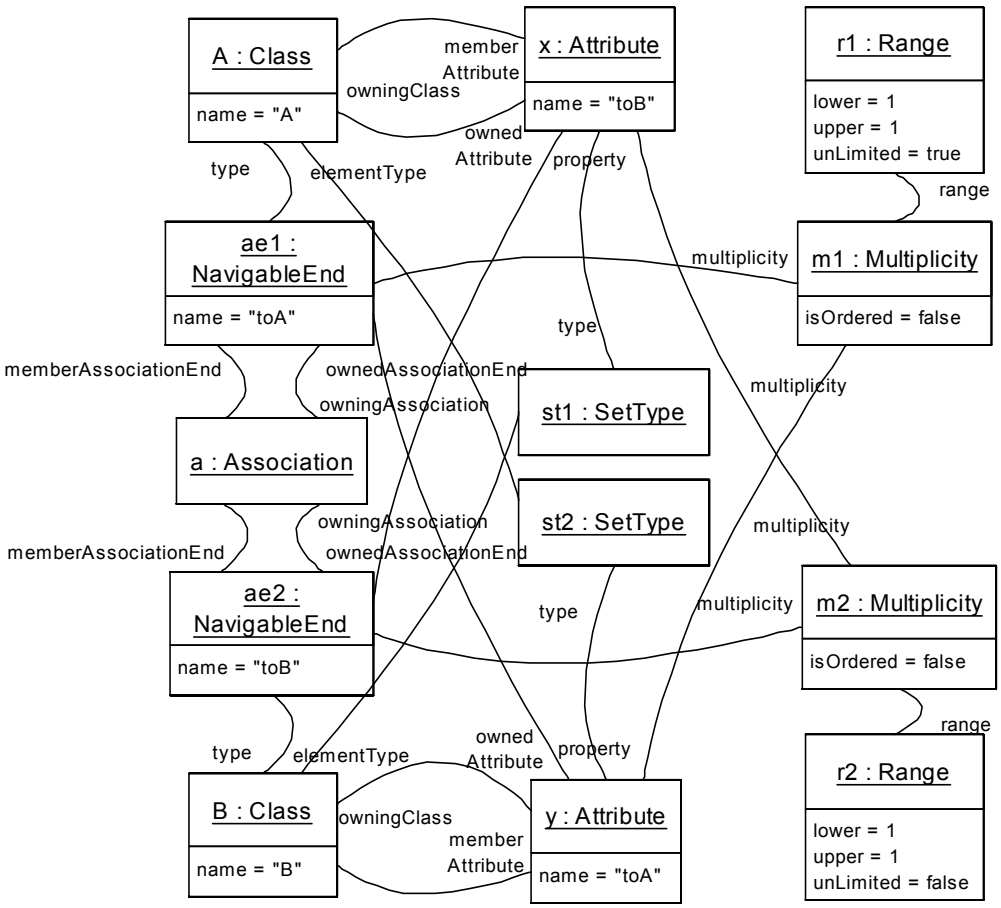


Figure 8-9 *Snapshot of Figure 8-8 on page 84*

Figure 8-10 on page 85 shows an example of a link and pair of navigable link ends that satisfy the properties of the above association. Note that each link end is associated with a slot through which an object can navigate to the objects at the opposite end of the link. Because the both association ends multiplicities are unordered, the appropriate slot values will be sets as opposed to sequences.

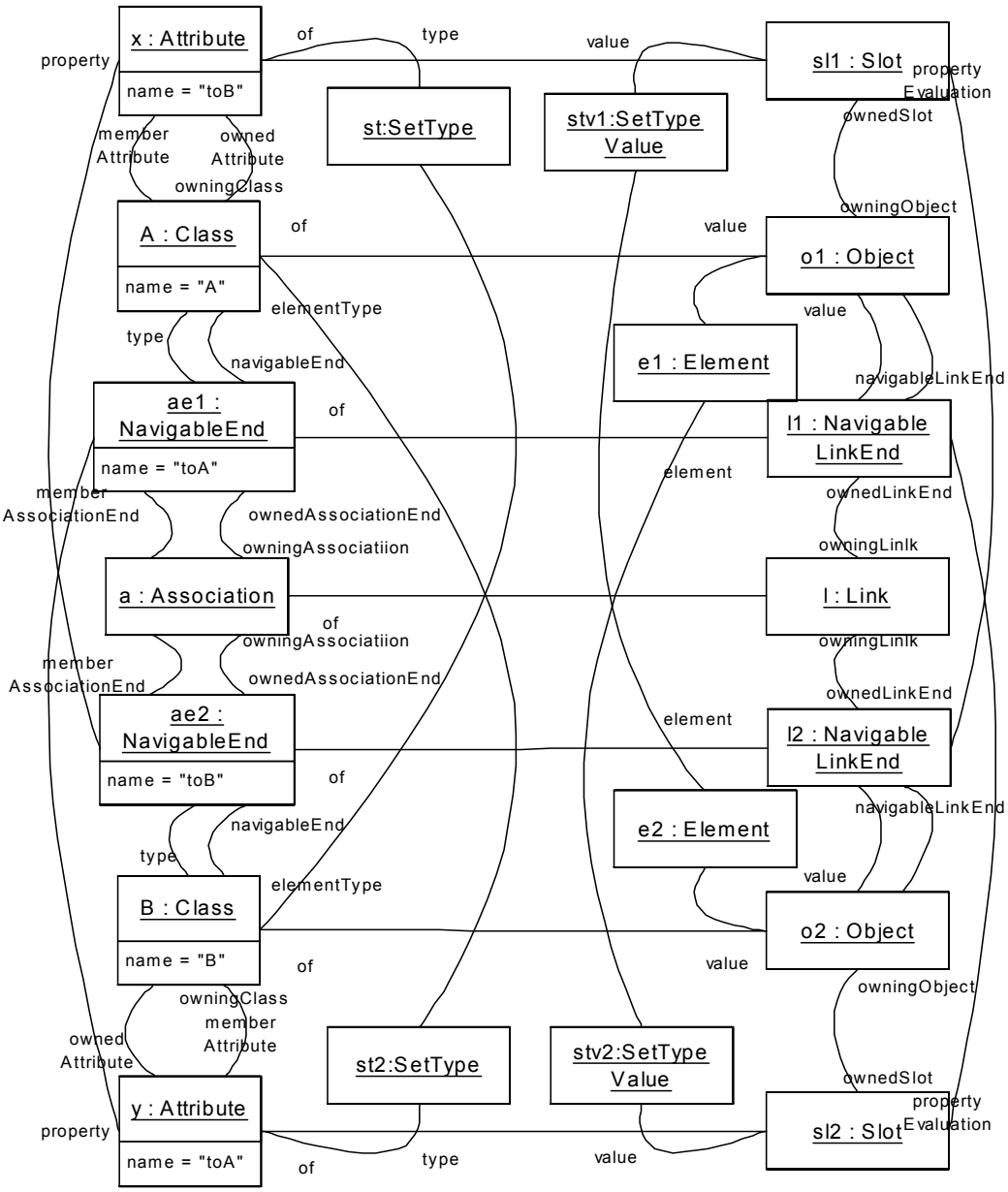


Figure 8-10 *Snapshot of Association Values*

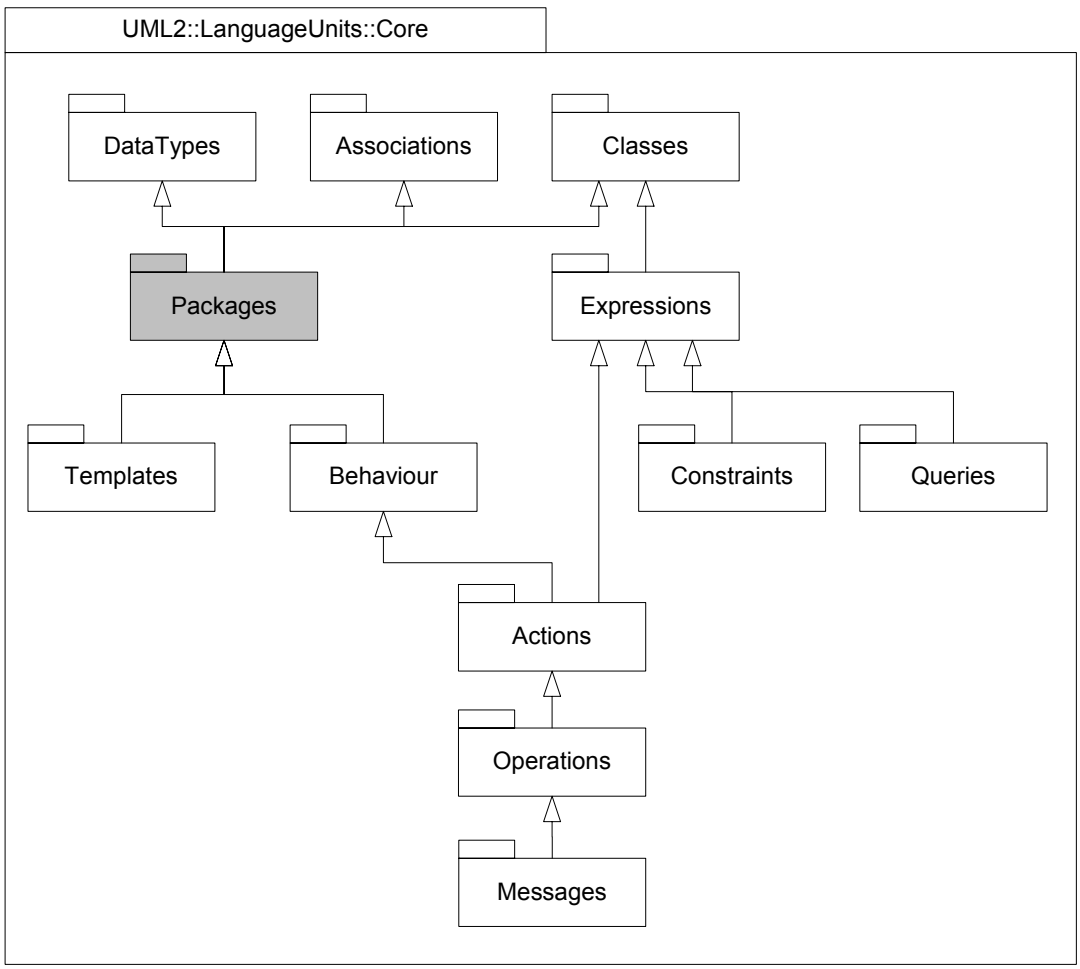
8.6 CHANGES FROM UML 1.4

Navigable link ends have been added and an explicit recognition that association ends can be interpreted as attributes or queries has been made.

Chapter 9
Packages

This package defines the abstract syntax and semantics of packages. Packages are namespaces for the elements they contain. Packages can also import elements into their namespace. This definition will be extended in Chapter 10, “Package Extension,” on page 93 with package extension mechanisms that will enable packages to be composed and reused in more sophisticated ways.

9.1 POSITION IN ARCHITECTURE



9.1.1 Example

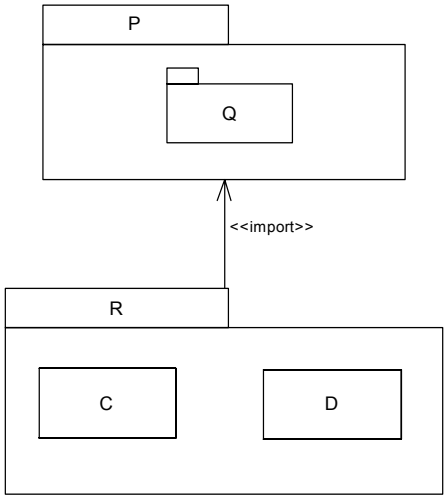


Figure 9-1 *An example of using Packages*

An example of the use of packages is shown in figure 9-1 on page 87. A package R contains two classes C and D. The package P containing a package Q is imported by R.

9.2 ABSTRACT SYNTAX

9.2.1 Derivation

Figure 9-2 on page 87 gives an overview of the templates used to stamp out using the Packages package. A Package is a namespace for named elements. A package may also import named elements from other packages. The named elements defined in the core are: classes, packages, associations and datatypes.

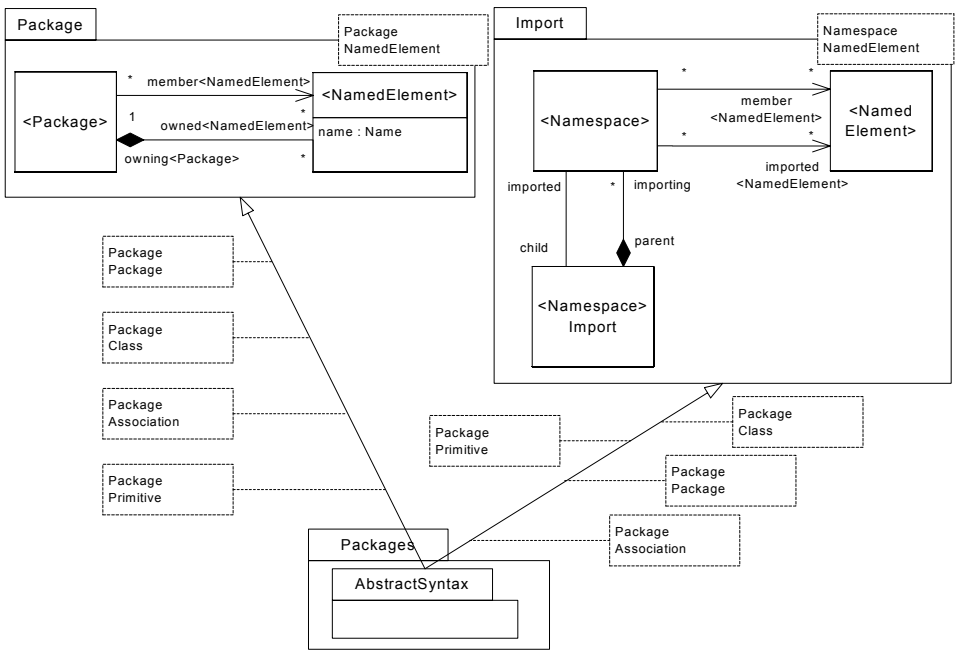


Figure 9-2 *Derivation of Packages*

9.2.2 Model

Figures 9-3 on page 88 show the abstract syntax of the Packages package. A package is a namespace for its classes, associations, packages and primitive datatypes. A package has owned elements, member elements and imported elements. Owned elements and imported elements are members of the namespace of a package.

A package imports all elements in the namespace of its imported packages into its own namespace. A package also imports all elements belonging to its containing package.

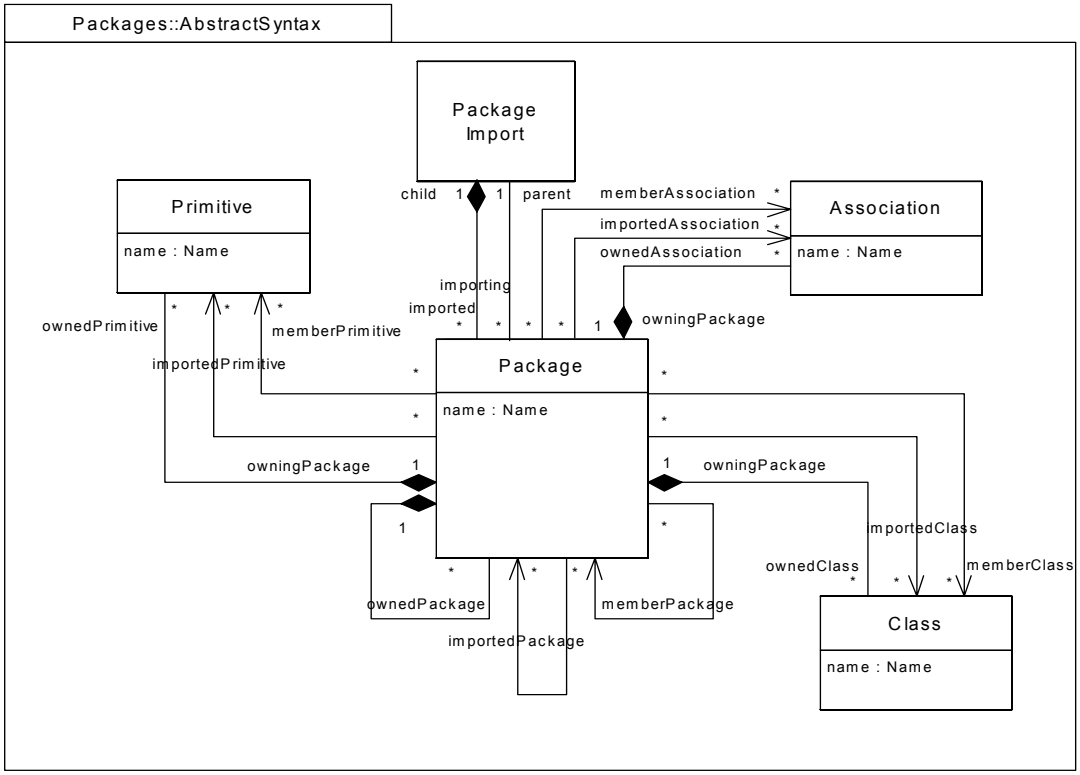


Figure 9-3 *Abstract Syntax for Packages package*

Package

A package is used to group related elements, and provides a namespace for those elements. Packages are also namespaces for their sub-packages.

Attributes

name The name of the package.

Associations

ownedAssociation The associations that are owned by the package.

importedAssociation The associations imported by the package.

memberAssociation The associations that are in the namespace of the package.

ownedClass The classes that are owned by the package.

importedClass The classes imported by the package.

memberClass The classes that are in the namespace of the package.

ownedPrimitive The primitive datatypes that are owned by the package.

memberPrimitive The primitive datatypes that are in the namespace of the package.

importedPrimitive The primitives imported by the package.

ownedPackage The packages that are owned by the package.

importedPackage The sub-packages imported by the package.

memberPackage The packages that are in the namespace of the package.

9.2.3 Well-formedness Rules

Package

[1] No two associations in a package's namespace may have the same name.

```
context Package inv
    self.memberAssociation -> forAll(e1 |
      self.memberAssociation -> forAll(e2 |
        e1 <> e2 implies e1.name <> e2.name))
```

[2] No two classes in a package's namespace may have the same name.

```
context Package inv
    self.memberClass -> forAll(e1 |
      self.memberClass -> forAll(e2 |
        e1 <> e2 implies e1.name <> e2.name))
```

[3] No two primitive datatypes in a package's namespace may have the same name.

```
context Package inv
    self.memberPrimitive -> forAll(e1 |
      self.memberPrimitive -> forAll(e2 |
        e1 <> e2 implies e1.name <> e2.name))
```

[4] No two packages in a package's namespace may have the same name.

```
context Package inv
    self.memberPackage -> forAll(e1 |
      self.memberPackage -> forAll(e2 |
        e1 <> e2 implies e1.name <> e2.name))
```

[5] Imported and owned associations, classes, primitives and packages belong to the namespace of the package.

```
context Package inv
   self.memberAssociation -> includesAll(self.importedAssociation ->
     union(self.ownedAssociation)) and
   self.memberClass -> includesAll(self.importedClass->union(self.ownedClass))
     and
   self.memberPrimitive -> includesAll(self.importedPrimitive->
     union(self.ownedPrimitive)) and
   self.memberPackage -> includesAll(self.importedPackage->
     union(self.ownedPackage))
```

[6] Imported associations, classes, primitives and packages cannot be owned and vice versa.

```
context Package inv
   self.importedAssociation -> intersection(self.ownedAssociation) -> isEmpty and
   self.importedClass -> intersection(self.ownedClass) -> isEmpty and
   self.importedPrimitive -> intersection(self.ownedPrimitive) -> isEmpty and
   self.importedPackage -> intersection(self.ownedPackage) -> isEmpty
```

[7] Parent packages associations are imported.

```
context Package inv:
  self.importedNamespaces()->forAll(x |
    self.importedAssociation->includesAll(x.memberAssociation))
```

[8] Parent packages classes are imported.

```
context Package inv:
  self.importedNamespaces()->forAll(x |
    self.importedClass->includesAll(x.memberClass))
```

[9] Parent packages primitives are imported.

```
context Package inv:
  self.importedNamespaces()->forAll(x |
    self.importedPrimitive->includesAll(x.memberPrimitive))
```

[10] Parent packages packages are imported.

```
context Package inv:
  self.importedNamespaces()->forAll(x |
    self.importedPackage->includesAll(x.memberPackage))
```

[11] A package imports its owning package's associations.

```
context Package inv
  self.owningPackage <> self implies
    self.memberClass->includesAll(self.owningPackage.memberAssociation)
```

[12] A package imports its owning package's classes.

```
context Package inv
  self.owningPackage <> self implies
self.memberClass->includesAll(self.owningPackage.memberClass)
```

[13] A package imports its owning package's packages.

```
context Package inv
  self.owningPackage <> self implies
    self.memberPackage->includesAll(self.owningPackage.memberPackage)
```

[14] A package imports its owning package's primitives.

```
context Package inv
  self.owningPackage <> self implies
    self.memberClass->includesAll(self.owningPackage.memberPrimitive)
```

9.2.4 Operations

Package

[1] Looks up an association in a package when given a name.

```
context Package::lookupAssociationforName(x : Name):Association
  self.memberAssociation->select(e| e.name = x ).selectElement()
```

[2] Looks up an association's name when given the association.

```
context Package::lookupNameForAssociation(x : Association):Name
  self.memberAssociation->select(e|e = x ).selectElement().name
```

[3] Looks up a class in a package when given a name.

```
context Package::lookupClassforName(x : Name):Class
  self.memberClass->select(e| e.name = x ).selectElement()
```

[4] Looks up a class's name when given the class.

```
context Package::lookupNameForClass(x : Class):Name
  self.memberClass->select(e|e = x ).selectElement().name
```

[5] Looks up a primitive in a package when given a name.

```
context Package::lookupPrimitiveforName(x : Name):Primitive
  self.memberPrimitive->select(e| e.name = x ).selectElement()
```

[6] Looks up a primitive's name when given the primitive.

```
context Package::lookupNameForPrimitive(x : Primitive):Name
  self.memberPrimitive->select(e|e = x ).selectElement().name
```

[7] Looks up a package in a package when given a name.

```
context Package::lookupPackageforName(x : Name):Package
  self.memberPackage->select(e| e.name = x ).selectElement()
```

[8] Looks up a package's name when given the package.

```
context Package::lookupNameForPackage(x : Package):Name
  self.memberPackage->select(e|e = x ).selectElement().name
```

[9] Returns the imported packages of the package.

```
context Package::importedPackage():Set(Package)
  self.imported->iterate(p s=Set{} | s->union(Set{p.parent}))
```

[10] Transitively returns all imported packages of the package.

```
context Package::allImportedPackage():Set(Package)
  self.importedPackage()->iterate(g s=self.importedPackage() |
         s->union(g.allImportedPackage()))
```

9.3 SEMANTIC DOMAIN

9.3.1 Derivation

The values in the packages package are derived from the PackageValue template shown in figure. A Package-Value is a container of named element values with identity.

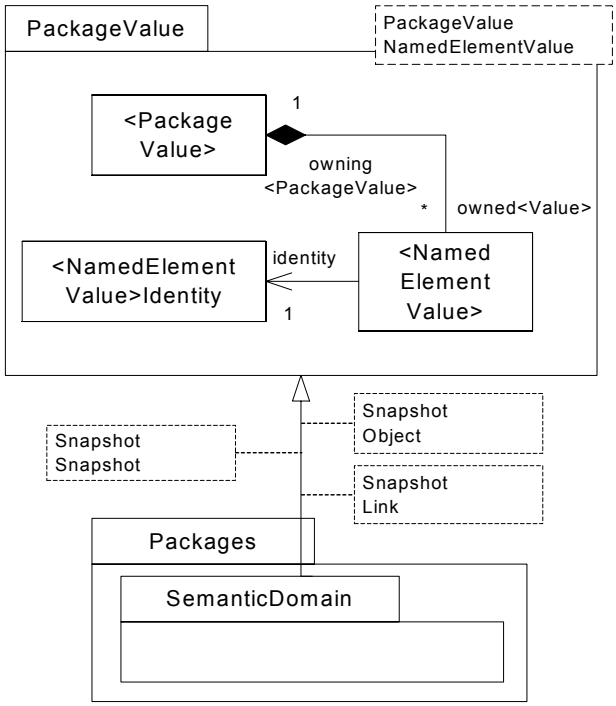


Figure 9-4 *Derivation of Packages Semantic Domain Package*

9.3.2 Model

The semantic domain of the Packages package is shown in 9-5 on page 93. A Snapshot is a value of a Package and describes a particular instantiation of the elements in the Package at a specific point in time. A Snapshot therefore contains objects, links, primitive values and snapshots. Objects, links and snapshots all have unique identities within a snapshot, whilst primitive values do not.

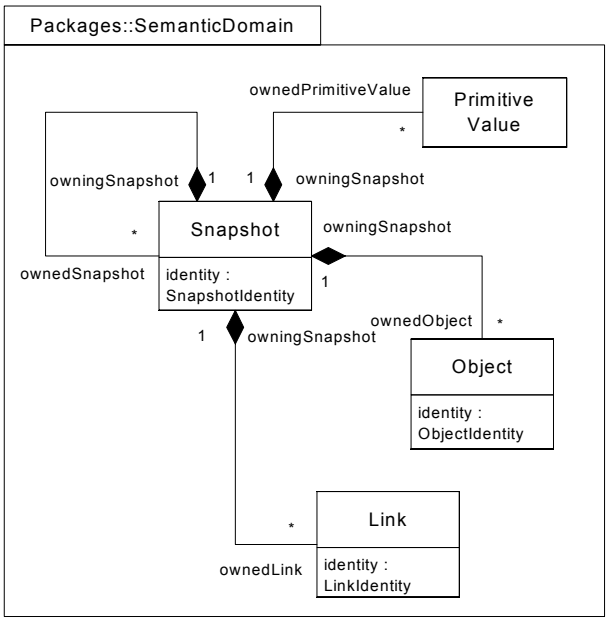


Figure 9-5 *Semantic Domain for the Packages package*

Snapshot

Snapshots are containers of objects, links, primitive values and snapshots.

Associations

ownedObject The objects owned by the snapshot.

ownedLink The links owned by the snapshot.

*ownedPrimitiveValue*The primitive values owned by the snapshot.

*ownedSnapshot*The snapshots owned by the snapshot.

9.3.3 Well-formedness rules

[1] No two objects in a snapshot's valuespace may have the same identity.

```
context Snapshot inv
    self.ownedObject -> forAll(e1 |
      self.ownedObject -> forAll(e2 |
        e1 <> e2 implies e1.identity <> e2.identity))
```

[2] No two links in snapshot's valuespace may have the same identity.

```
context Snapshot inv
    self.ownedLink -> forAll(e1 |
      self.ownedLink -> forAll(e2 |
        e1 <> e2 implies e1.identity <> e2.identity))
```

[3] No two snapshots in snapshot's valuespace may have the same identity.

```
context Snapshot inv
    self.ownedSnapshot -> forAll(e1 |
      self.ownedSnapshot -> forAll(e2 |
        e1 <> e2 implies e1.identity <> e2.identity))
```

9.4 SEMANTIC MAPPING

9.4.1 Derivation

The template used to stamp out the semantic mapping for the packages package is shown in figure 9-6 on page 94. Each element in the semantic domain is mapped to the appropriate abstract syntax element and the necessary constraints on their relationships are stamped out.

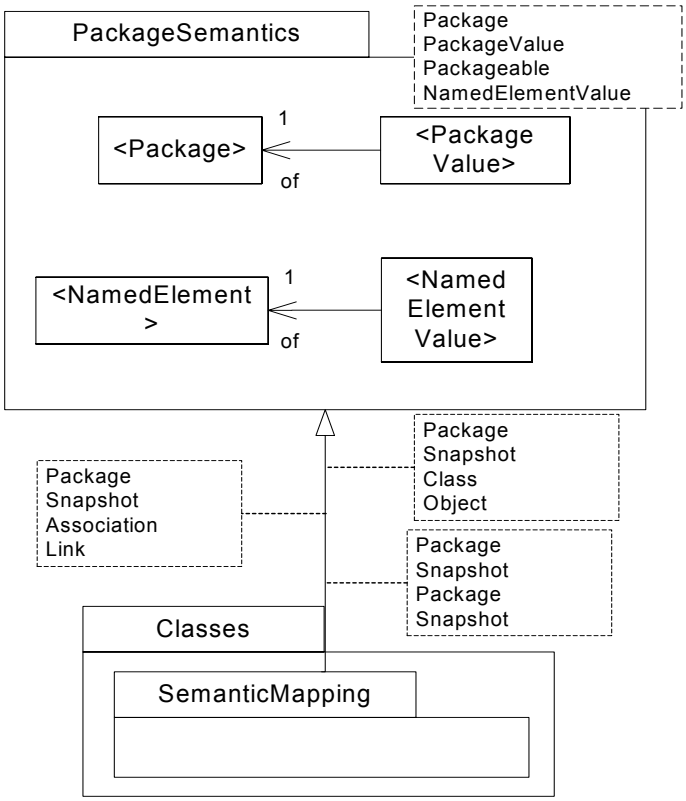


Figure 9-6 *Derivation of the Packages SemanticMapping Package*

9.4.2 Model

The semantics mapping package of the packages package is shown in Figure 9-7 on page 95. It defines the relationship that holds between packages, named elements and their values. A Snapshot is a value of a Package. An Object is a value of an Class. A Link is a value of an Association and a primitive value is a value of a primitive

data type. The objects contained by a snapshot must be values of the classes owned by the snapshot's package, and similarly for the other values..

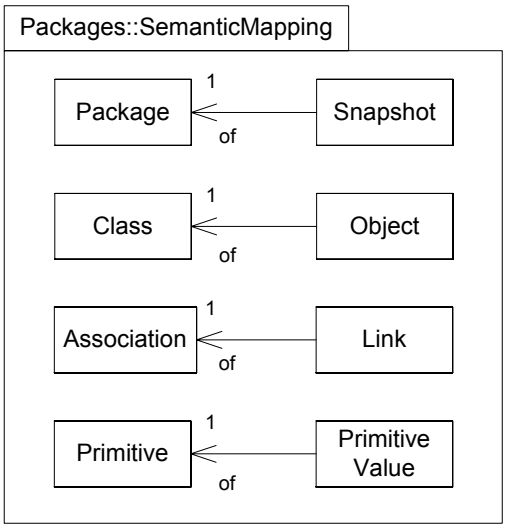


Figure 9-7 *Semantic Mapping for the Packages package*

9.4.3 Well-formedness rules

Snapshot

[1] For each object owned by a snapshot there should be a class of the snapshot's package's namespace that the object is a value of.

```
context Snapshot inv:
  self.ownedObject->forAll(c |
    self.of.memberClass->exists(d | c.of = d))
```

[2] For each link owned by a snapshot there should be an association of the snapshot's package's namespace that the link is a value of.

```
context Snapshot inv:
  self.ownedLink->forAll(c |
    self.of.memberAssociation->exists(d | c.of = d))
```

[3] For each primitive value owned by a snapshot there should be a primitive of the snapshot's package's namespace that the primitive value is a value of.

```
context Snapshot inv:
  self.ownedPrimitiveValue->forAll(c |
    self.of.memberPrimitive->exists(d | c.of = d))
```

[4] For each snapshot owned by a snapshot there should be a package of the snapshot's package's namespace that the snapshot is a value of.

```
context Snapshot inv:
  self.ownedSnapshot->forAll(c |
    self.of.memberPackage->exists(d | c.of = d))
```

9.5 EXAMPLE SNAPSHOTS

Figure 9-9 on page 96 illustrates a snapshot corresponding to the model shown in 9-8 on page 96. Note how the package import results in an import relationship between package R and the contents of package P (i.e. R imports P::Q into its namespace).

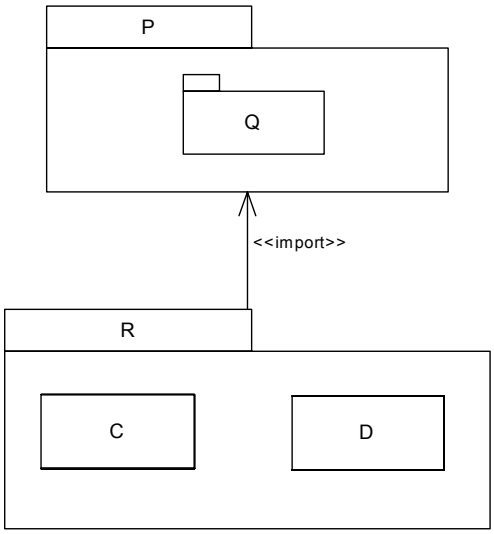


Figure 9-8 *Example packages*

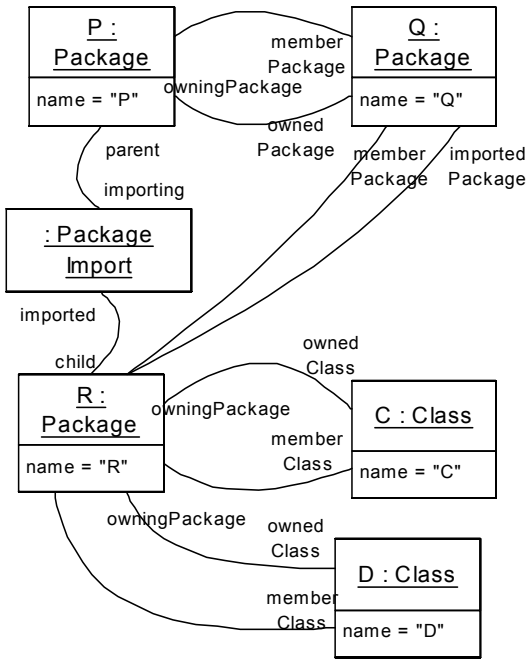


Figure 9-9 *Snapshot of example shown in fig. 9-8 on page 96*

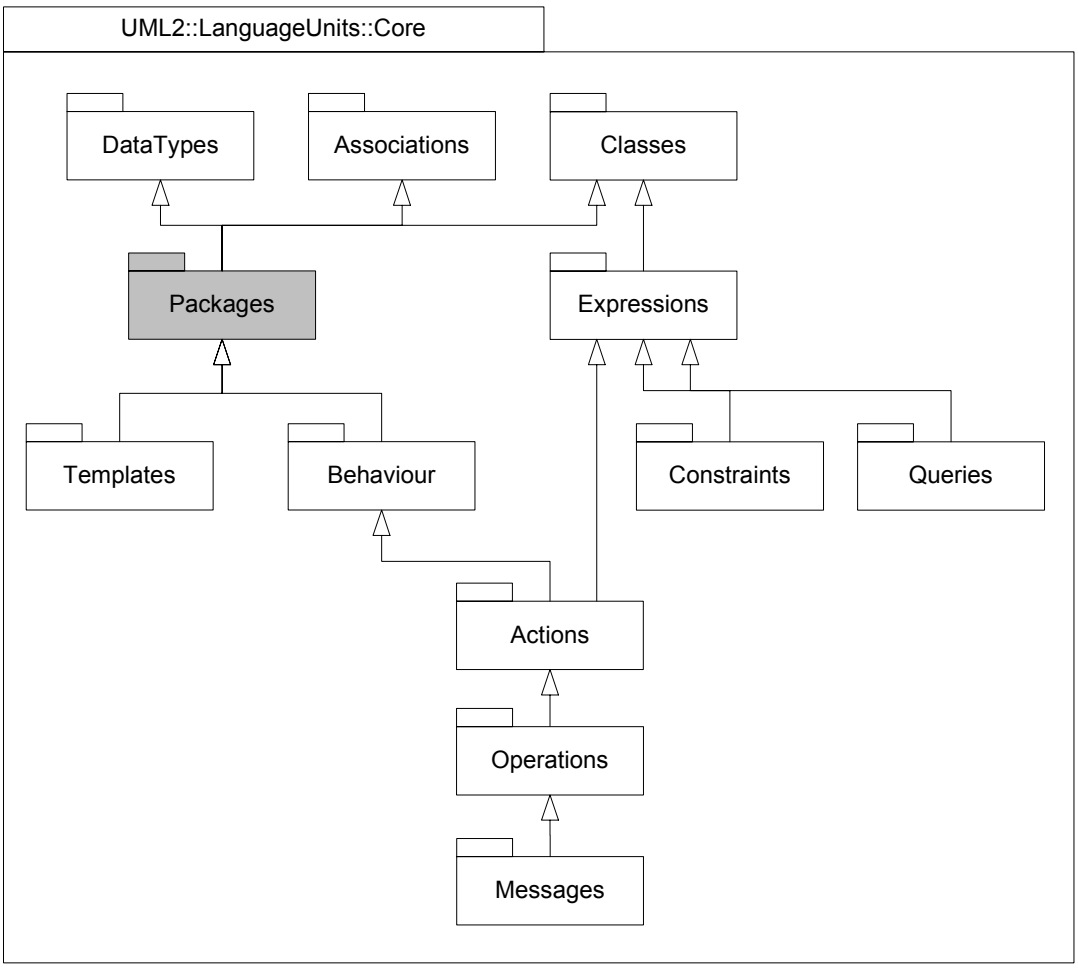
9.6 CHANGES TO UML 1.4

Packages have values (snapshots). Snapshots are an extremely useful abstraction for modelling system level states. They will be extended in later chapters to deal with dynamic aspects (filmstrips).

Chapter 10
Package Extension

This package defines an extended abstract syntax and semantics for packages that permits their use as a powerful "aspect-oriented" extension mechanism. In their most basic form, packages are namespaces for the elements they contain. In the definition presented in this chapter, packages can additionally extend other packages, extending, renaming and merging their elements. The ability to reuse large-grained language components through package extension is a fundamental part of this submission.

10.1 Position in Architecture



10.1.1 Example

Figure 10-1 on page 99 illustrates the use of package extension to merge and extend the contents of two packages P and Q. Becuase the class A in Q is redefined during extension, the end result (shown in grey) is to merge the contents of the two classes A and B into a single class A in R.

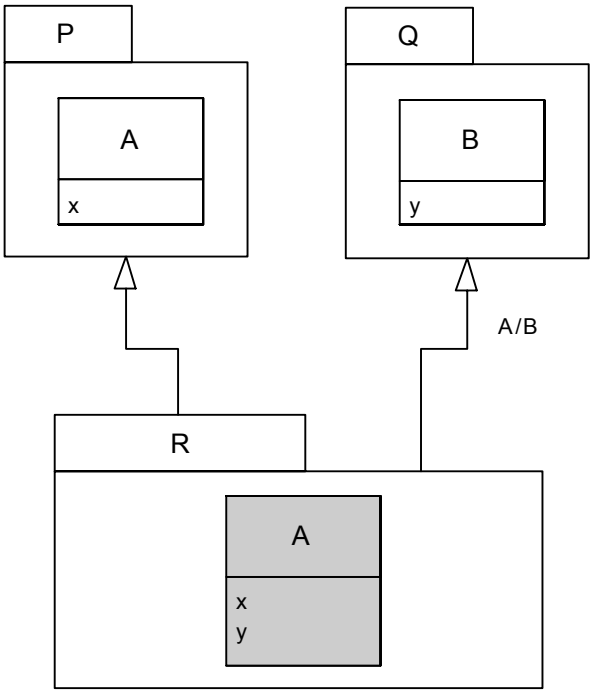


Figure 10-1 *Example of package extension*

10.2 ABSTRACT SYNTAX

10.2.1 Derivation

Figure 10-3 on page 101 gives an overview of the templates used to stamp out the extensions part of the Packages package. Templates are used to generate extension relationships between all namespace and feature elements in the core, including packages, classes, associations, association ends, attributes and operations.

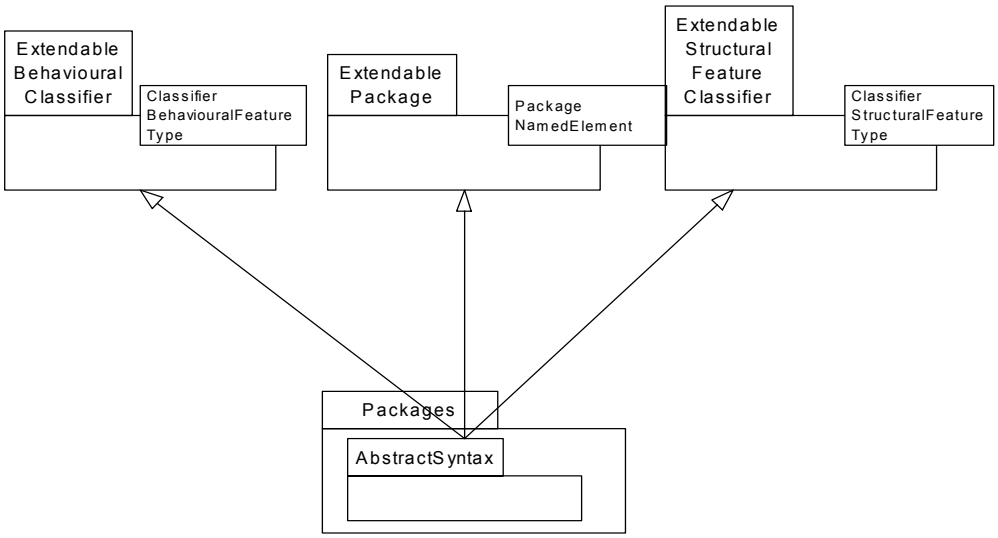


Figure 10-2 *Extension Templates*

The ExtendablePackage template (see Figure 10-3 on page 101) describes the notion of package extension. When a package extends another package, the elements in the parent package's namespace are extended into the namespace of the child package. For example, an element may be a class or an association. Extending a package will result in the classes and association in the namespace of the parent package/s being extended into the child package's namespace. Note that the definition is deliberately abstract about how this is implemented: for example an element may be inherited or copied - the choice of mechanism is entirely up to the implementor. However, in the case where an element is redefined, it must copied down (see [Clark02]). A redefined extension represents an explicit substitution of one element by another.

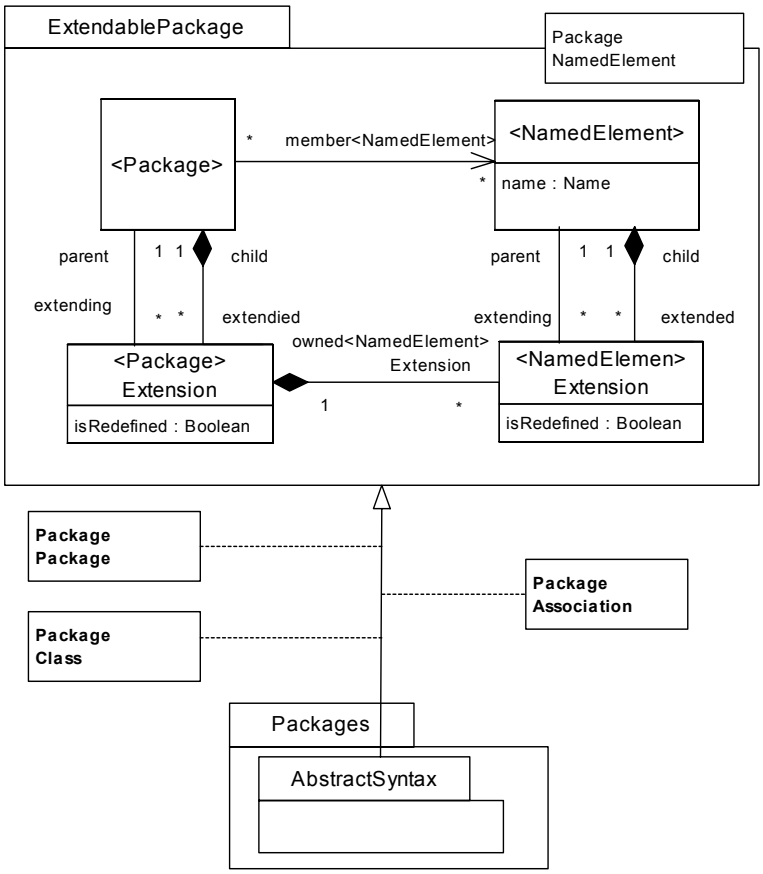


Figure 10-3 *Derivation of Packages from extendable package templates*

The ExtendableStructuralFeatureClassifier template (see Figure 10-4 on page 102) defines the semantics of classifier and structural feature extension. When a classifier extends another classifier, the structural features in the parent classifier/s namespace are extended into the namespace of the child classifier. For example, extending a class will result in the class's attributes being extended into the namespace of the extending class. In addition, a structural feature that is extended into a namespace must be conformant with the structural feature it extends, for example their types must be conformant. If a redefinition has occurred, the child structural feature's type must also belong to the same namespace as the child class.

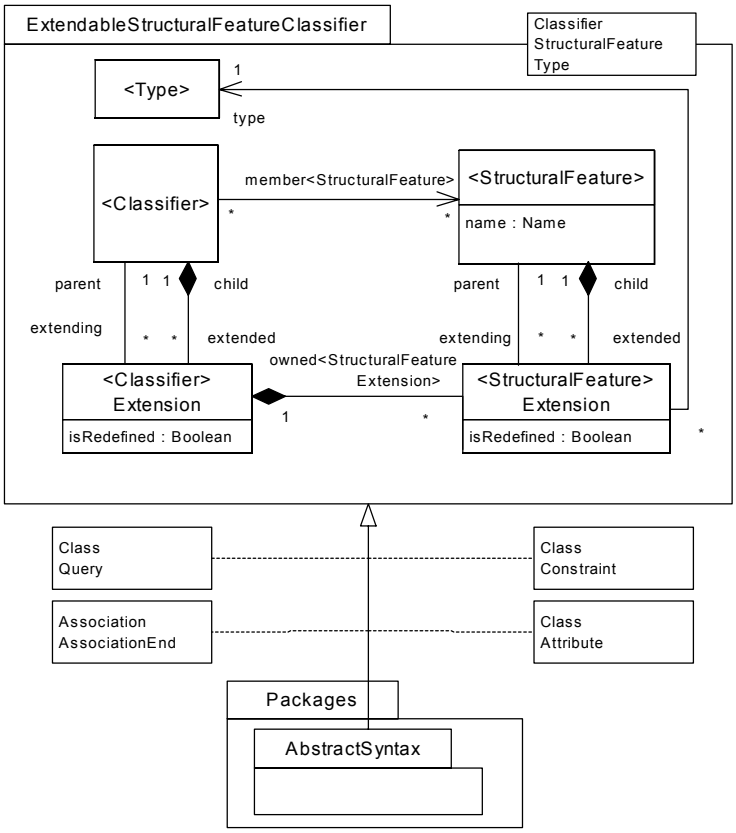


Figure 10-4 *Derivation of Packages from ExtendableStructuralFeatureClassifier template*

The ExtendableBehaviouralFeature template (see Figure 10-5 on page 103) describes the general extension relationship between classifiers and their behavioural features. A classifier can extend another classifier with the result that the parent's behavioural features are extended into the namespace of the child classifier. It is also required that an extended behavioural feature's type and parameters conform to the type and parameters of its parent behavioural feature. If a renaming or redefinition has occurred, the child behavioural feature's types must belong to the same namespace as the behavioural feature.

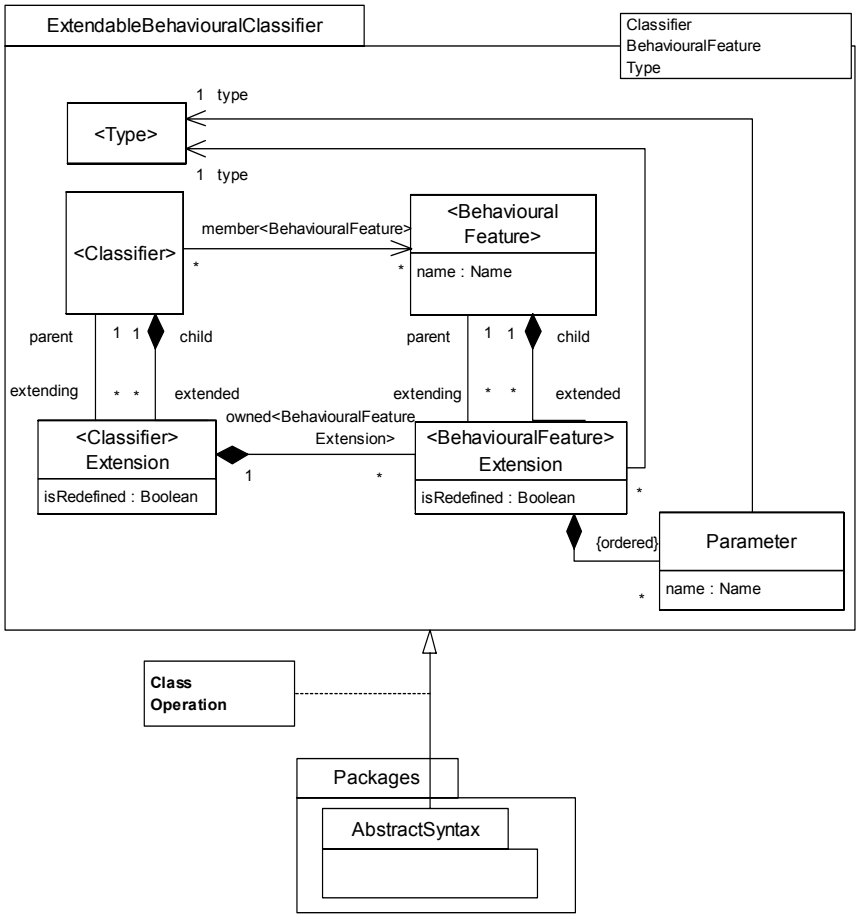


Figure 10-5 *Derivation of Packages from ExtendableBehaviouralFeature template*

10.2.2 Model (Package extension)

Figures 10-6 on page 104 to Figure 10-8 on page 111 show the abstract syntax of the extensions part of the Packages package. As shown in Figures 10-6 on page 104 packages that extend packages will include extended classes, associations and sub-packages as a part of their namespace. Extensions can be redefined, which means that no restriction is placed on the names of the child elements in the relationship.

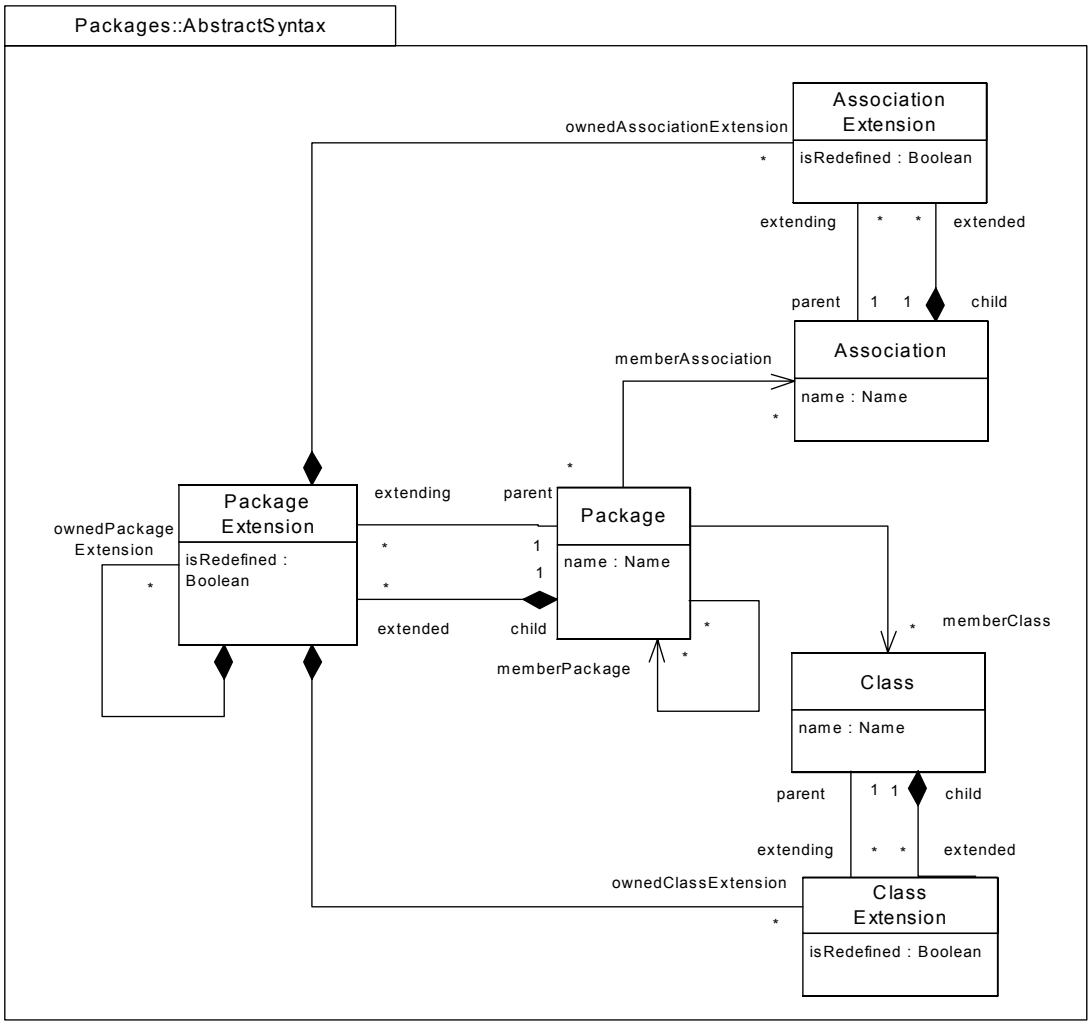


Figure 10-6 *Abstract syntax for Packages package*

Package

A package.

Associations

memberAssociation The associations that are included in the namespace of the package.

memberClass The classes that are included in the namespace of the package.

memberPackage The packages that are included in the namespace of the package.

PackageExtension

An extension relationship between packages. When a package extends another package, the parent packages elements are included in the namespace of the child package. A package extension has a set of renamings that are applied to any elements copied from the parent package to the child package.

Associations

child The child package.

ownedAssociationExtension The association extensions that extend associations in the parent packages namespace.

ownedClassExtension The class extensions that extend classes in the parent packages namespace.

ownedPackageExtension The package extensions that extend packages in the parent packages namespace.

parent The parent package.

renaming The renamings that apply to elements extended from the parent package's namespace.

10.2.3 Well-formedness Rules (Package extension)

PackageExtension

[1] The associations in the namespace of the parent package must be included in the namespace of the child and they must be related by an association extension.

```
context PackageExtension inv:
  self.parent.memberAssociation->forAll(e |
    self.ownedAssociationExtension->exists(e' |
      e'.parent = e and
      self.child.memberAssociation->exists(e'' |
        e'.child = e'')))
```

[2] If the child association does not equal the parent association in an ownedAssociationExtension then it must be owned by the child package.

```
context PackageExtension inv:
  self.ownedAssociationExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedAssociation -> includes(e.child))
```

[3] The classes in the namespace of the parent package must be included in the namespace of the child and they must be related by a class extension.

```
context PackageExtension inv:
  self.parent.memberClass->forAll(e |
    self.ownedClassExtension->exists(e' |
      e'.parent = e and
      self.child.memberClass->exists(e'' |
        e'.child = e'')))
```

[4] If the child class does not equal the parent class in an ownedClassExtension then it must be owned by the child package.

```
context PackageExtension inv:
  self.ownedClassExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedClass -> includes(e.child))
```

[5] The packages in the namespace of the parent package must be included in the namespace of the child and they must be related by a package extension.

```
context PackageExtension inv:
  self.parent.memberPackage->forAll(e |
    self.ownedPackageExtension->exists(e' |
      e'.parent = e and
      self.child.memberPackage->exists(e'' |
        e'.child = e'')))
```

[6] If the child package does not equal the parent package in an ownedPackageExtension then it must be owned by the child package.

```
context PackageExtension inv:
  self.ownedPackageExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedPackage -> includes(e.child))
```

[7] The child package must have the same name as the parent, unless it is redefined..

```
context PackageExtension inv:
  not self.isRedefined implies child.name = parent.name
```

10.2.4 Model (Structural features)

As shown in Figures 10-7 on page 107 classes that extend classes will include (extended) attributes, constraints and queries as a part of their namespace. Associations that extend associations include (extended) association ends. Again, extensions can be redefined, which means that no restriction is placed on the names of the child elements in the relationship.

AssociationExtension

An extension relationship between associations. An association extension has a set of renamings that are applied to extended association ends.

Associations

child The child association.

ownedAssociationEndExtension The association end extensions that extend association ends in the parent associations namespace.

parent The parent association.

renaming The renamings that apply to association ends extended from the parent association.

AssociationEndExtension

An extension relationship between association ends.

Associations

child The child association end.

parent The parent association end.

AttributeExtension

An extension relationship between attributes.

Associations

child The child attribute.

parent The parent attribute.

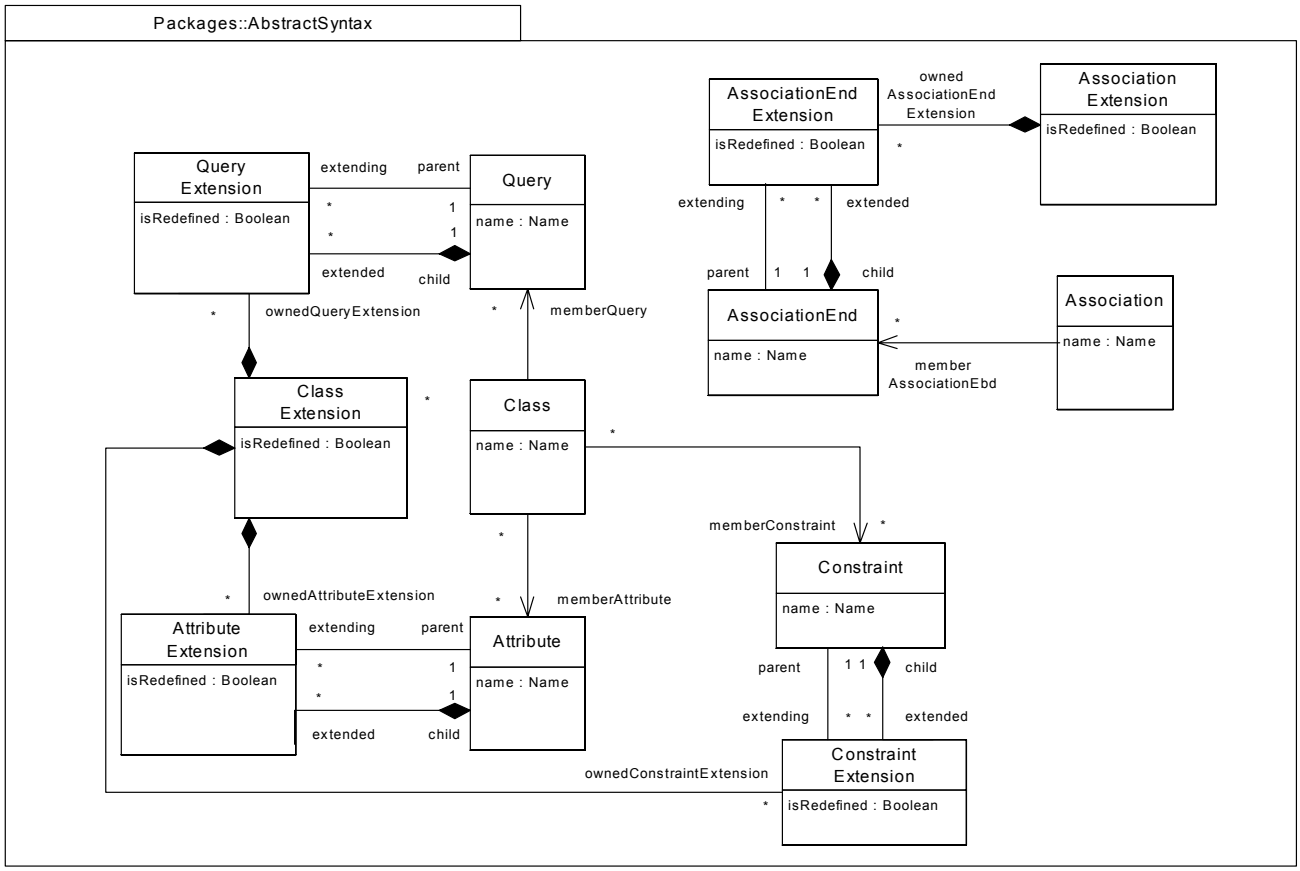


Figure 10-7 *Abstract syntax for the Package package*

ClassExtension

An extension relationship between classes. A class extension has a set of renamings that are applied to any elements copied from the parent class to the child class.

Associations

child The child class.

ownedAttributeExtension The attribute extensions that extend attributes in the parent classes namespace.

ownedConstraintExtension The constraint extensions that extend constraints in the parent classes namespace.

ownedQueryExtension The query extensions that extend queries in the parent classes namespace.

parent The parent class.

renaming The renamings that apply to any element copied from the parent class.

ConstraintExtension

An extension relationship between constraints.

Associations

child The child constraint.

parent The parent constraint.

QueryExtension

An extension relationship between queries.

Associations

child The child query.

parent The parent query.

10.2.5 Well-formedness Rules (Structural features)

AssociationExtension

[1] The association ends in the namespace of the parent association must be included in the namespace of the child association and they must be related by an association end extension.

```
context AssociationExtension inv:
  self.parent.memberAssociationEnd->forAll(e |
    self.ownedAssociationEndExtension->exists(e' |
      e'.parent = e and
      self.child.memberAssociationEnd->exists(e'' |
        e'.child = e'')))
```

[2] If the child association end doesn't equal the parent association end in an ownedAssociationEndExtension then it must be owned by the child association.

```
context AssociationExtension inv:
  self.ownedAssociationEndExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedAssociationEnd -> includes(e.child))
```

[3] The child association must have the same name as the parent association, unless it is redefined.

```
context AssociationExtension inv:
  not self.isRedefined implies child.name = parent.name
```

AssociationEndExtension

[1] The child association end's type in an association end extension must conform to the parent association end's type.

```
context AssociationEndExtension inv:
  self.child.type.conformsToExtension(self.parent.type)
```

[2] The child association end's multiplicity in an association end extension must conform to the parent association end's multiplicity.

```
context AssociationEndExtension inv:
  self.child.multiplicity.conformsToExtension(self.parent.multiplicity)
```

[3] If the child association end in an association end extension has been extended into another namespace (i.e. the child does not equal the parent) then the child's type must belong to the same namespace as the child's class.

```
context AttributeExtension inv:
 self.child <> self.parent implies
   self.child.owningClass.sameNamespace(self.child.type)
```

[4] The child association end must have the same name as the parent association end, unless it is redefined..

```
context AssociationEndExtension inv:
 not self.isRedefined implies child.name = parent.name
```

AttributeExtension

[1] The child attribute's type in an attribute extension must conform the parent attribute's type.

```
context AttributeExtension inv:
  self.child.type.conformsToExtension(self.parent.type)
```

[2] The child attribute's multiplicity in an attribute extension must conform the parent attribute's multiplicity.

```
context AttributeExtension inv:
  self.parent.multiplicity <> null implies
    self.child.multiplicity.conformsToExtension(self.parent.multiplicity)
```

[3] If the child attribute in an attribute extension has been extended into another namespace (i.e. the child does not equal the parent) then the child's type must belong to the same namespace as the child's class.

```
context AttributeExtension inv:
 self.child <> self.parent implies
   self.child.owningClass.sameNamespace(self.child.type)
```

[4] The child attribute must have the same name as the parent attribute, unless it is redefined..

```
context AttributeExtension inv:
  not self.isRedefined implies child.name = parent.name
```

ClassExtension

[1] The attributes in the namespace of the parent class must be included in the namespace of the child class and they must be related by an attribute extension.

```
context ClassExtension inv:
  self.parent.memberAttribute->forAll(e |
    self.ownedAttributeExtension->exists(e' |
      e'.parent = e and
      self.child.memberAttribute->exists(e'' |
        e'.child = e'')))
```

[2] If the child attribute does not equal the parent attribute in an ownedAttributeExtension then it must be owned by the child class.

```
context ClassExtension inv:
  self.ownedAttributeExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedAttribute -> includes(e.child))
```

[3] The constraints in the namespace of the parent class must be included in the namespace of the child class and they must be related by a constraint extension.

```
context ClassExtension inv:
  self.parent.memberConstraint->forAll(e |
    self.ownedConstraintExtension->exists(e' |
      e'.parent = e and
      self.child.memberConstraint->exists(e'' |
        e'.child = e'')))
```

[4] If the child constraint does not equal the parent constraint in an ownedConstraintExtension then it must be owned by the child class.

```
context ClassExtension inv:
  self.ownedConstraintExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedConstraint -> includes(e.child))
```

[5] The queries in the namespace of the parent class must be included in the namespace of the child class and they must be related by a query extension.

```
context ClassExtension inv:
  self.parent.memberQuery->forAll(e |
    self.ownedQueyExtension->exists(e' |
      e'.parent = e and
      self.child.memberQuery->exists(e'' |
        e'.child = e'')))
```

[6] If the child query doesn't equal the parent query in an ownedQueryExtension then it must be owned by the child class.

```
context ClassExtension inv:
  self.ownedQueryExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedQuery -> includes(e.child))
```

ConstraintExtension

[1] The child constraint's type in an constraint extension must conform to the parent constraint's type.

```
context ConstraintExtension inv:
  self.child.type.conformsToExtension(self.parent.type)
```

[2] The child constraint's expression in an constraint extension must conform to the parent constraint's expression.

```
context ConstraintExtension inv:
  self.child.expression.conformsToExtension(self.parent.expression)
```

[3] If the child constraint in an constraint extension has been extended into another namespace (i.e. the child does not equal the parent) then the child's type must be in the same namespace as the child's class.

```
context ConstraintExtension inv:
 self.child <> self.parent implies
   self.child.owningClass.sameNamespace(self.child.type)
```

[4] The child constraint must have the same name as the parent constraint, unless it is redefined..

```
context AttributeExtension inv:
  not self.isRedefined implies child.name = parent.name
```

QueryExtension

[1] The child query's type in an query extension must conform to the parent query's type.

```
context QueryExtension inv:
  self.child.type.conformsToExtension(self.parent.type)
```

[2] The child query's expression in an query extension must conform to the parent query's expression.

```
context QueryExtension inv:
  self.child.expression.conformsToExtension(self.parent.expression)
```

[3] The child query must have the same name as the parent query, unless it is redefined..

```
context QueryExtension inv:
 not self.isRedefined implies child.name = parent.name
```

[4] If the child query in a query extension has been extended into another namespace (i.e. the child does not equal the parent) then the child's type must belong to the same namespace as the child's class.

```
context QueryExtension inv:
  self.child <> self.parent implies
    self.child.owningClass.sameNamespace(self.child.type)
```

10.2.6 Model (Behavioural features)

As shown in Figures 10-8 on page 111 classes that extend classes will include operations as a part of their namespace. Extensions can be redefined, which means that no restriction is placed on the names of the child elements in the relationship.

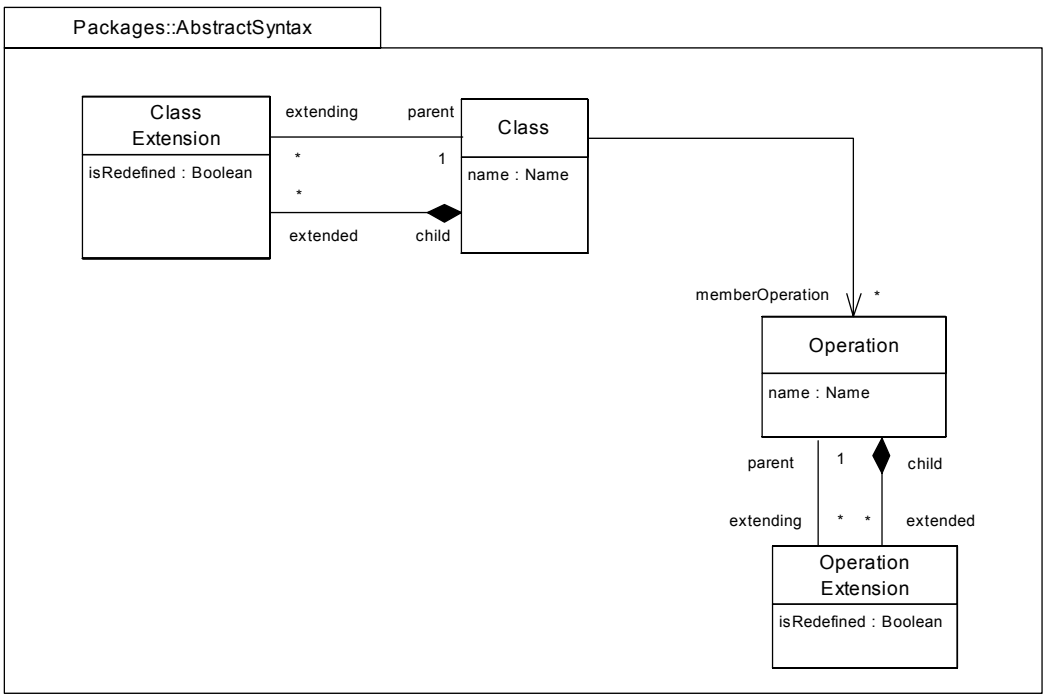


Figure 10-8 *Abstract Syntax for the Packages package*

OperationExtension

An extension relationship between operations.

Associations

child The child operation.

parent The parent operation.

10.2.7 Well-formedness Rules (Behavioural features)

ClassExtension

[1] The actions in the namespace of the parent class must be included in the namespace of the child class and they must be related by an action extension.

```
context ClassExtension inv:
  self.parent.memberAction->forAll(e |
```

```
      self.ownedActionExtension->exists(e' |
        e'.parent = e and
        self.child.memberAction->exists(e'' |
          e'.child = e'')))
```

[2] If the child action doesn't equal the parent action in an ownedActionExtension then it must be owned by the child class.

```
context ClassExtension inv:
  self.ownedActionExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedAction -> includes(e.child))
```

[3] The operations in the namespace of the parent class must be included in the namespace of the child class and they must be related by an operation extension.

```
context ClassExtension inv:
  self.parent.memberOperation->forAll(e |
    self.ownedOperationExtension->exists(e' |
      e'.parent = e and
      self.child.memberOperation->exists(e'' |
        e'.child = e'')))
```

[4] If the child operation doesn't equal the parent operation in an ownedOperationExtension then it must be owned by the child class.

```
context ClassExtension inv:
  self.ownedOperationExtension -> forAll(e |
    e.child <> e.parent implies
      self.child.ownedOperation -> includes(e.child))
```

OperationExtension

[1] The child operation;s type in an operation extension must conform to the parent operation's type.

```
context OperationExtension inv:
   self.child.type.conformsToExtension(self.parent.type)
```

[2] The child operation's parameter types must be an extension of the parent parent's parameter types.

```
context OperationExtension inv:
  self.parent.parameter -> forAll(f |
    1..(self.child.parameter->size) -> forAll(n |
      self.child.parameter.at(n).type.conformsToExtension(
        f.parameter.at(n).type)))
```

[3] If the child operation in an operation extension has been extended into another namespace (i.e. the child does not equal the parent) then the child's types must be in the same namespace as the child's class.

```
context OperationExtension inv:
  self.child <> self.parent implies
    self.child.owningClass.sameNamespace(self.child.type) and
    self.child.parameter -> forAll(f |
      self.child.owningClass.sameNamespace(f)
```

[4] The child operation must have the same name as the parent operation, unless it is redefined..

```
context OperationExtension inv:
  not self.isRedefined implies child.name = parent.name
```

10.2.8 Additional Definitions

A number of additional definitions are required to support the extension mechanism. Firstly, the conformsToExtension() operation must be defined on the data types. The most important is for a class:

Class

[1] A class conforms to another class if its is extended.

```
context Class::conformsToExtension(c : Class) : Boolean
  self.allExtendedElements()->includes(c)
```

and similarly for the other datatypes.

Secondly, conformance rules for multiplicities needs to be defined:

Multiplicity

[1] A multiplicity conforms to another multiplicity if their ranges are conformant.

```
context Multiplicity::conformsToExtension(m : Multiplicity) : Boolean
  TBD
```

Finally, conformance rules for expressions needs to be defined:

Expression

[1] An expression conforms to another expression if they are conformant extensions.

```
context Expression::conformsToExtension(m : Expression) : Boolean
  TBD
```

This will be defined recursively, considering each kind of expression in turn. The aim is to check that the expression conforms to the expression passed as argument, and that the sub-expressions, where present, also conform, and so on.

10.3 SEMANTIC DOMAIN

No additional semantics.

10.4 SEMANTIC MAPPING

No additional semantics.

10.5 EXAMPLE SNAPSHOTS

Fugure 10-10 on page 114 shows the example package extension model shown in figure 10-9 on page 114 as a snapshot. Note that the redefinition of class B in package Q, permits it to have a different name, i.e. A. Because two classes with the same name are extending into package R, they must be merged to be well-formed. The result is to also merge their contents (i.e. attributes).

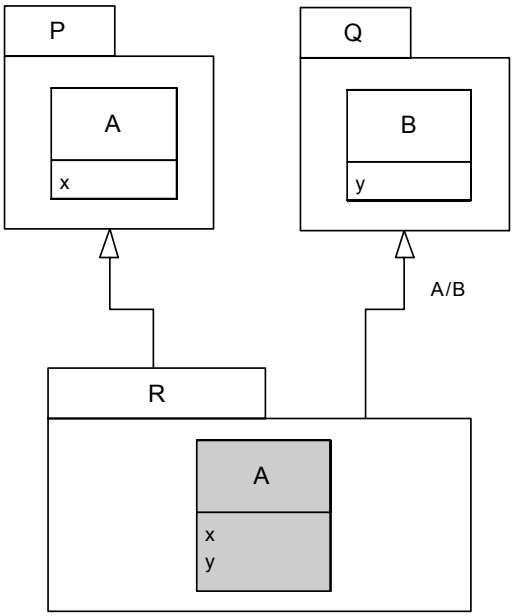


Figure 10-9 *Example of package extension*

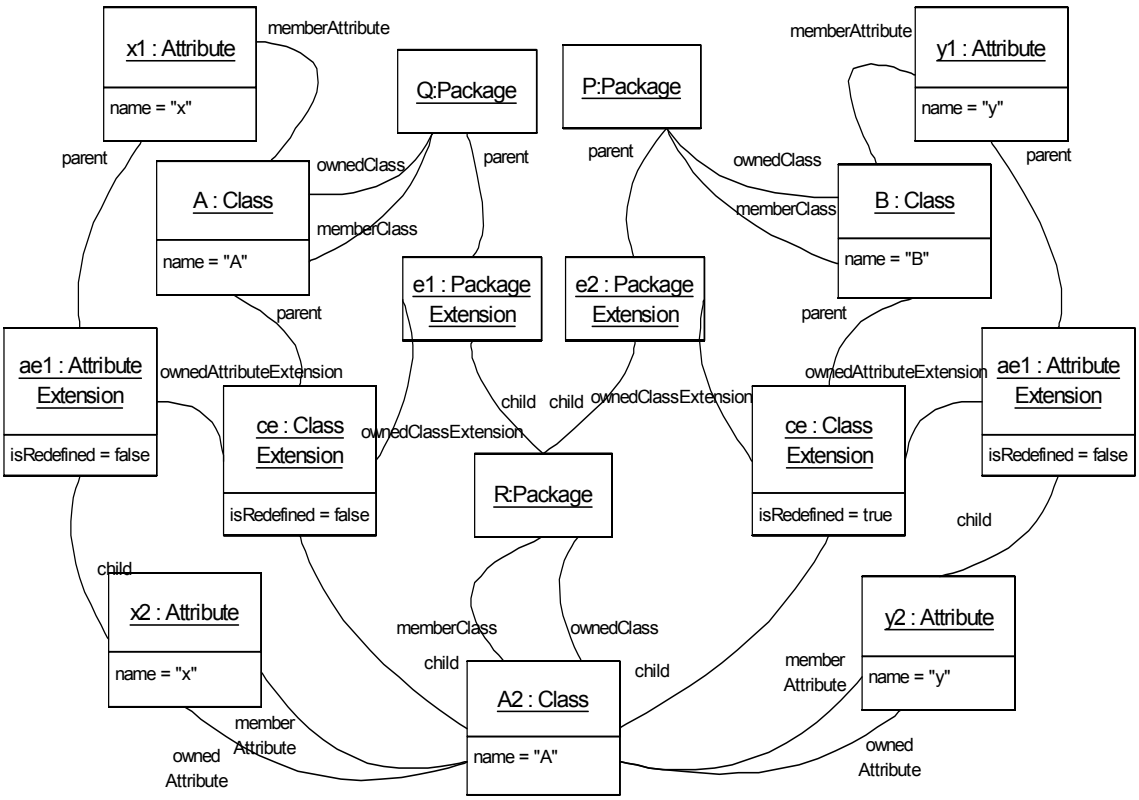


Figure 10-10 *Snapshot of Figure 10-9 on page 114*

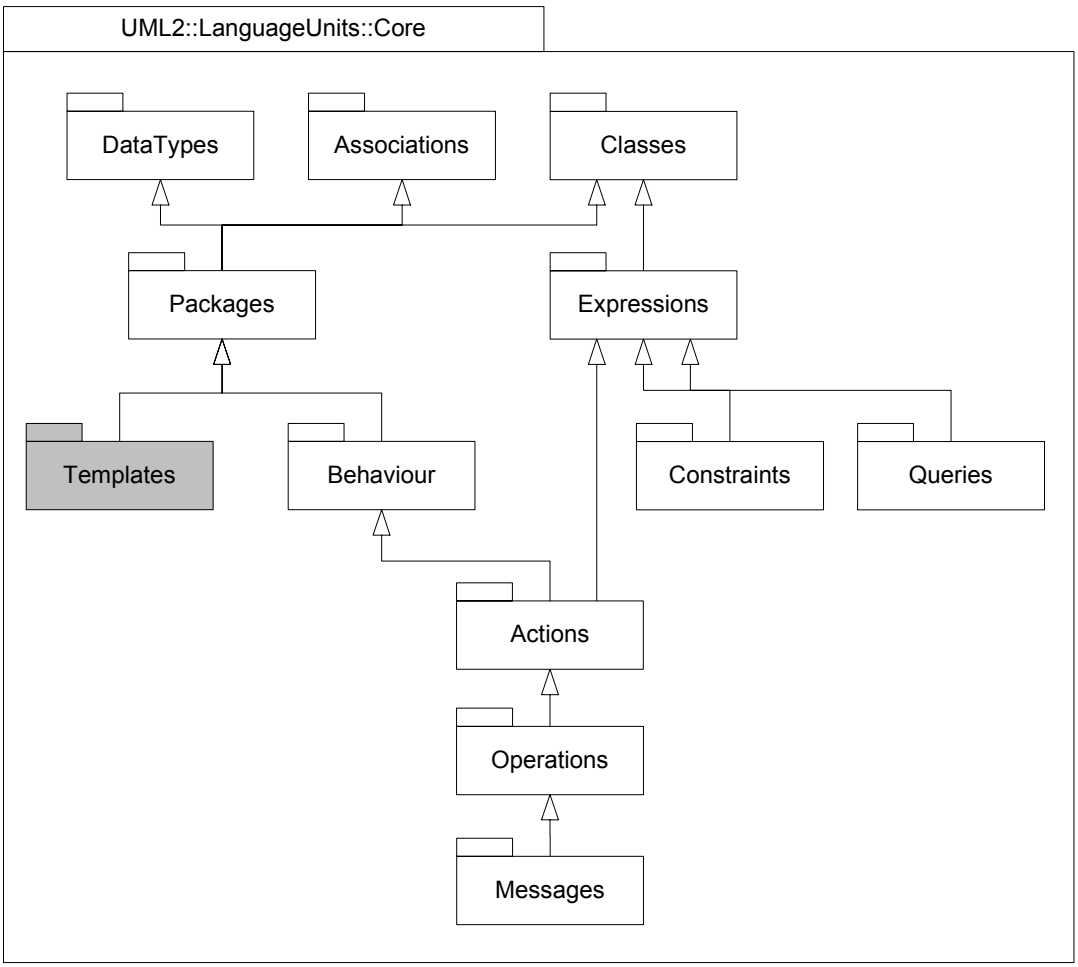
10.6 CHANGES TO UML 1.4

Package extension is new to UML 2.0.

Chapter 11
Templates

A package template is an extendable package with substitutable parameter variables. In this chapter, the definition of packages and package extension is extended to support package templates. A description of class templates and association templates is also given to illustrate the generic nature of the template used in the definition.

11.1 Position in Architecture



11.1.1 Example

As shown below a package template is a namespace for named elements, whose names can be placeholders for parameters passed by the package template. Package template instantiation is an extension relationship between a

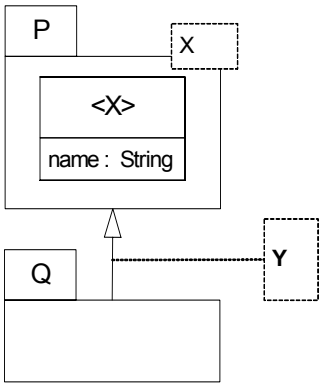


Figure 11-1 *Example package template*

package template and package, in which substitutions can be made for the parameters. In this example, the value Y is substituted for X, resulting in the class <X> being copied and renamed to "Y" in the package Q.

11.2 ABSTRACT SYNTAX

11.2.1 Derivation

Figure 11-2 on page 118 shows the templates used to derive the abstract syntax and well-formedness rules for package templates. A template is a namespace for name elements which may have a renaming expression attached to them. For example, a package template may attach a renaming expression "<X>" to a class. A TemplateInstantiation defines an instantiation relationship that generates named element extensions with the appropriate name substitutions.

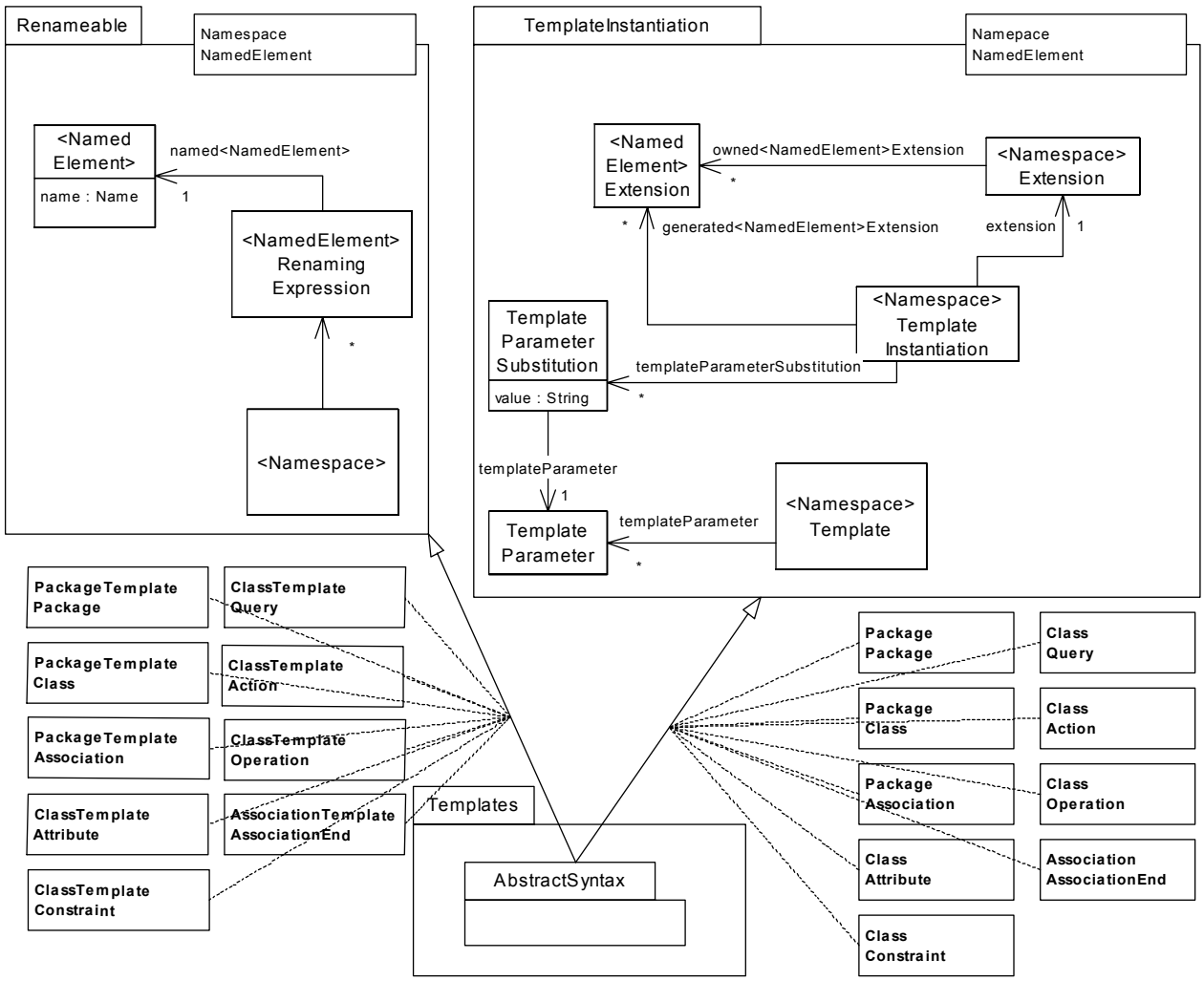


Figure 11-2 *Derivation of the Templates Package*

11.2.2 Model

Figure 11-3 on page 118 extends a RenamingExpression so that it can describe a simple renaming expression language (similar to that used in this submission), including parameterised values, e.g. "<X>" and concatenated values, e.g. "owned<X>".

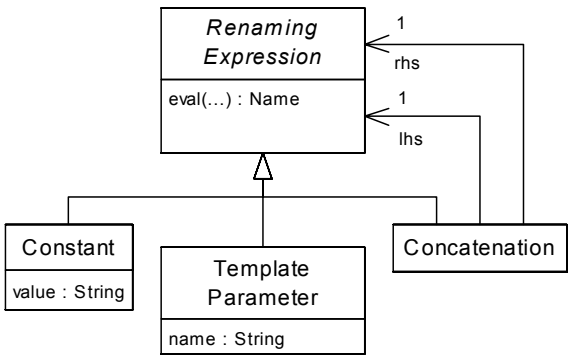


Figure 11-3 *Renaming Expressions*

Figure 11-4 on page 119 shows the abstract syntax of the templates package that describes package templates. A PackageTemplate is a Package and therefore can be extended as described in the Package Extension chapter. A PackageTemplate owns a set of template parameters and a set of renaming expressions.

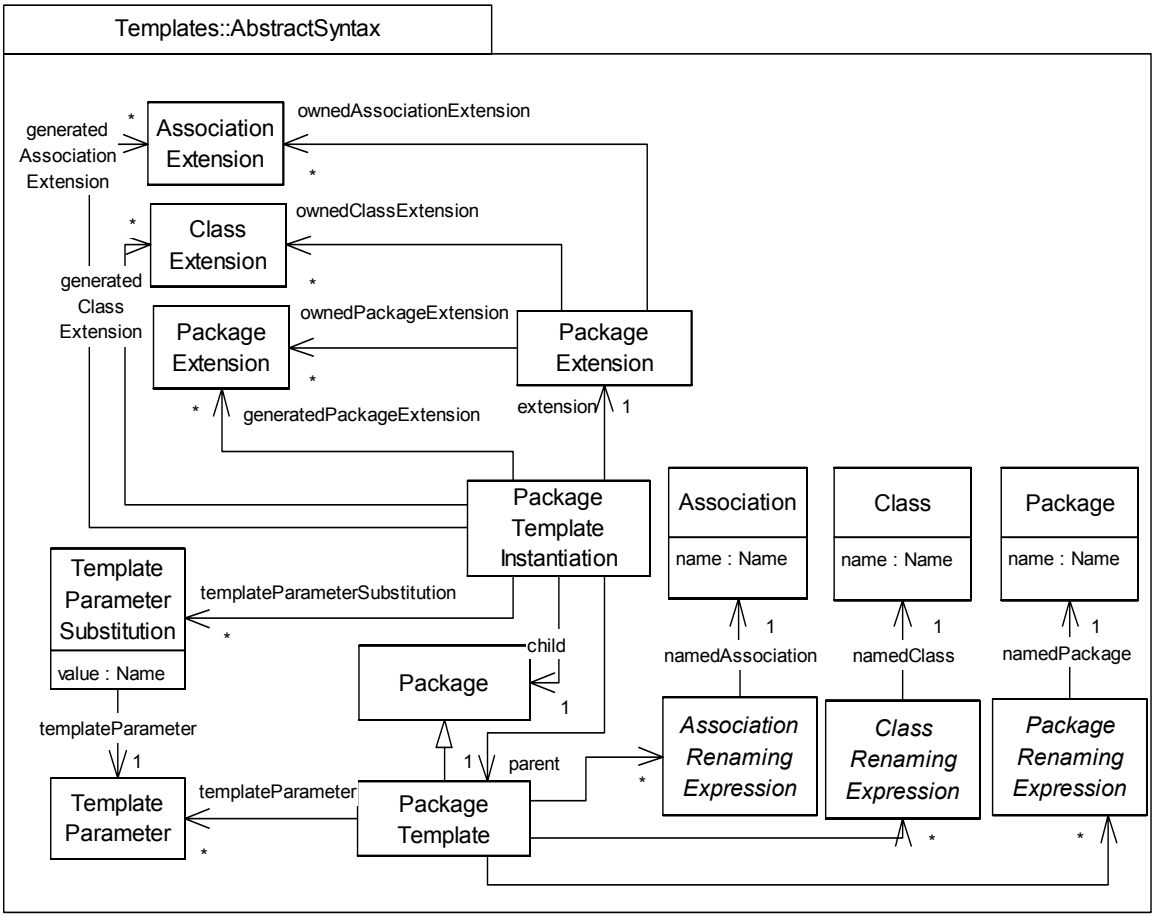


Figure 11-4 *Templates Abstract Syntax package (package templates)*

PackageTemplate

A package template.

Associations

renamingExpression The renaming expressions that are associated with the contents of the package template.

templateParameter The parameters of the package template.

PackageTemplateInstantiation

An instantiation relationship between a package template and a package.

Associations

child The package that results from the instantiation.

parent The package template.

templateParameterSubstitution The parameters that are substituted when instantiating the template.

generatedAssociationExtension The association extensions that are generated to realise the instantiation.

generatedClassExtension The class extensions that are generated to realise the instantiation.

generatedPackageExtension The package extensions that are generated to realise the instantiation

TemplateParameter

A template parameter. A subclass of RenamingExpression.

TemplateParameterSubstitution

The substitution that is made for a template parameter.

Attributes

value The value that is being substituted.

Associations

templateParameter The parameter that is being substituted for.

11.2.3 Well-formedness Rules

A number of rules are necessary to ensure that a PackageTemplateInstantiation is well-formed. The most important of these are as follows. Firstly, a PackageTemplateInstantiation must guarantee to rename all parameters in its parent TemplatePackage. Secondly, redefined association, class and package extensions must be generated for each of the substitutions that take place in the instantiation.

PackageTemplate

[1] Only one renaming expression per association in a template.

```
context PackageTemplate inv:
  self.associationRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedAssociation <> r2.namedAssociation)
```

[2] Only associations in the template's namespace have renaming expressions associated with them.

```
context PackageTemplate inv:
  self.memberAssociation->
    includesAll(self.associationRenamingExpression.namedAssociation)
```

[3] Only one renaming expression per class in a template.

```
context PackageTemplate inv:
  self.classRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedClass <> r2.namedClass)
```

[4] Only classes in the template's namespace have renaming expressions associated with them.

```
context PackageTemplate inv:
  self.memberClass->
    includesAll(self.classRenamingExpression.namedClass)
```

[5] Only one renaming expression per package in a template.

```
context PackageTemplate inv:
  self.packageRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedPackage <> r2.namedPackage)
```

[6] Only packages in the template's namespace have renaming expressions associated with them.

```
context PackageTemplate inv:
  self.memberPackage->
    includesAll(self.packageRenamingExpression.namedClass)
```

PackageTemplateInstantiaton

[1] Parameter substitutions parameters must match those owned by the template.

```
context PackageTemplateInstantiation inv:
  self.templateParameterSubstitutions.templateParameter =
  self.ownedParameter->asBag
```

[2] Association substitutions are generated for each of the renamed associations in the parents namespace.

```
context PackageTemplateInstantiation inv:
  self.generatedAssociationExtension.parent =
  self.extension.parent.associationRenamingExpression.namedAssociation
```

[3] Generated association extensions shadow redefined owned associations extensions.

```
context PackageTemplateInstantiation inv:
  self.extension.ownedAssociationExtension->select(e | e.isRedefined) =
  self.generatedAssociationExtension
```

[4] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context PackageTemplateInstantiation inv:
  self.generatedAssociationExtension->forAll(n |
    n.child.name = self.associationRenamingExpression->
      select(r | r.namedAssociation = n.parent).eval(self)->asSet)
```

[5] Class substitutions are generated for each of the renamed classes in the parents namespace.

```
context PackageTemplateInstantiation inv:
  self.generatedClassExtension.parent =
  self.extension.parent.classRenamingExpression.namedClass
```

[6] Generated class extensions shadow redefined owned class extensions.

```
context PackageTemplateInstantiation inv:
  self.extension.ownedClassExtension->select(e | e.isRedefined) =
  self.generatedClassExtension
```

[7] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context PackageTemplateInstantiation inv:
  self.generatedClassExtension->forAll(n |
    n.child.name = self.classRenamingExpression->
      select(r | r.namedClass = n.parent).eval(self)->asSet)
```

[8] Package substitutions are generated for each of the renamed packages in the parents namespace.

```
context PackageTemplateInstantiation inv:
  self.generatedPackageExtension.parent =
  self.extension.parent.packageRenamingExpression.namedPackage
```

[9] Generated package extensions shadow redefined owned package extensions.

```
context PackageTemplateInstantiation inv:
  self.extension.ownedPackageExtension->select(e | e.isRedefined) =
  self.generatedPackageExtension
```

[10] The name of the child elements of any generated package extension is the evaluation of the appropriate renaming expression.

```
context PackageTemplateInstantiation inv:
  self.generatedPackageExtension->forAll(n |
    n.child.name = self.packageRenamingExpression->
      select(r | r.namedPackage = n.parent).eval(self)->asSet)
```

Figure 11-5 on page 122 shows the abstract syntax of the templates package that describes class templates. A ClassTemplate is a Class and therefore can be extended as described in the Package Extension chapter. A TemplateClass owns a set of template parameters and a set of renaming expressions.

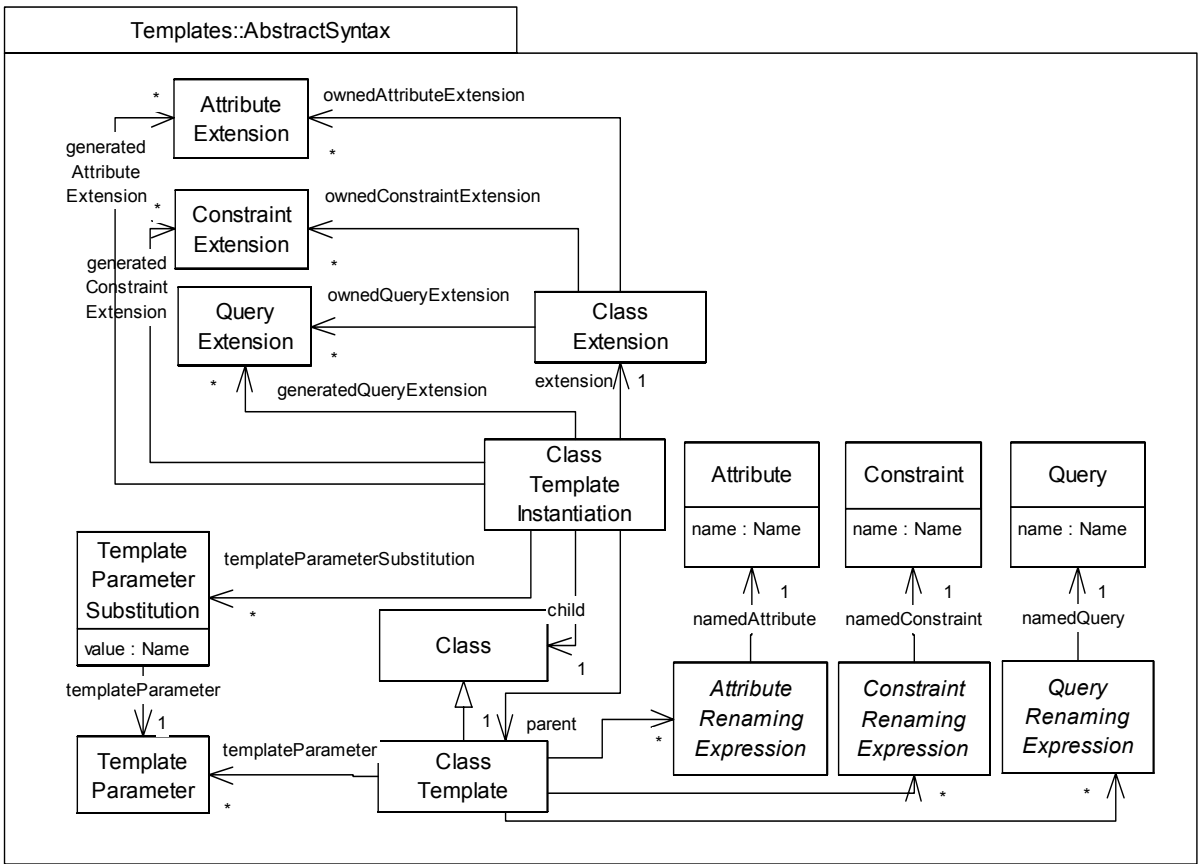


Figure 11-5 *Templates Abstract Syntax package (class templates)*

ClassTemplate

A class template.

Associations

renamingExpression The renaming expressions that are associated with the contents of the class template.

templateParameter The parameters of the class template.

ClassTemplateInstantiation

An instantiation relationship between a class template and a class.

Associations

child The package that results from the instantiation.

parent The package template.

templateParameterSubstitution The parameters that are substituted when instantiating the template.

generatedAttributeExtension The attribute extensions that are generated to realise the instantiation.

generatedConstraintExtension The constraint extensions that are generated to realise the instantiation.

generatedQueryExtension The constraint extensions that are generated to realise the instantiation.

11.2.4 Well-formedness Rules

A number of rules are necessary to ensure that a ClassTemplateInstantiation is well-formed. These are similar to those defined for PackageTemplateInstantiation. A ClassTemplateInstantiation must guarantee to rename all parameters in its parent ClassTemplate. Secondly, redefined attribute, constraint and query extensions must be generated for each of the substitutions that take place in the instantiation.

ClassTemplate

[1] Only one renaming expression per attribute in a template.

```
context ClassTemplate inv:
  self.attributeRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedAttribute <> r2.namedAttribute)
```

[2] Only attributes in the template's namespace have renaming expressions associated with them.

```
context ClassTemplate inv:
  self.memberAttribute->
    includesAll(self.attributeRenamingExpression.namedAttribute)
```

[3] Only one renaming expression per constraint in a template.

```
context ClassTemplate inv:
  self.constraintRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedConstraint <> r2.namedConstraint)
```

[4] Only attributes in the template's namespace have renaming expressions associated with them.

```
context ClassTemplate inv:
  self.memberConstraint->
    includesAll(self.constraintRenamingExpression.namedConstraint)
```

[5] Only one renaming expression per query in a template.

```
context ClassTemplate inv:
  self.queryRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedQuery <> r2.namedQuery)
```

[6] Only packages in the template's namespace have renaming expressions associated with them.

```
context ClassTemplate inv:
  self.memberQuery->
    includesAll(self.queryRenamingExpression.namedQuery)
```

ClassTemplateInstantiaton

[1] Parameter substitutions parameters must match those owned by the template.

```
context ClassTemplateInstantiation inv:
  self.templateParameterSubstitutions.templateParameter =
  self.ownedParameter->asBag
```

[2] Attribute substitutions are generated for each of the renamed classes in the parents namespace.

```
context ClassTemplateInstantiation inv:
  self.generatedAttributeExtension.parent =
  self.extension.parent.attributeRenamingExpression.namedAttribute
```

[3] Generated attribute extensions shadow redefined owned attribute extensions.

```
context ClassTemplateInstantiation inv:
  self.extension.ownedAttributeExtension->select(e | e.isRedefined) =
  self.generatedAttributeExtension
```

[4] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context ClassTemplateInstantiation inv:
  self.generatedAttributeExtension->forAll(n |
    n.child.name = self.attributeRenamingExpression->
      select(r | r.namedAttribute = n.parent).eval(self)->asSet)
```

[5] Constraint substitutions are generated for each of the renamed constraints in the parents namespace.

```
context ClassTemplateInstantiation inv:
  self.generatedConstraintExtension.parent =
  self.extension.parent.constraintRenamingExpression.namedConstraint
```

[6] Generated constraint extensions shadow redefined owned constraint extensions.

```
context ClassTemplateInstantiation inv:
  self.extension.ownedConstraintExtension->select(e | e.isRedefined) =
  self.generatedConstraintExtension
```

[7] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context ClassTemplateInstantiation inv:
  self.generatedConstraintExtension->forAll(n |
    n.child.name = self.constraintRenamingExpression->
      select(r | r.namedConstraint = n.parent).eval(self)->asSet)
```

[8] Query substitutions are generated for each of the renamed queries in the parents namespace.

```
context ClassTemplateInstantiation inv:
  self.generatedQueryExtension.parent =
  self.extension.parent.queryRenamingExpression.namedQuery
```

[9] Generated query extensions shadow redefined owned query extensions.

```
context ClassTemplateInstantiation inv:
  self.extension.ownedQueryExtension->select(e | e.isRedefined) =
  self.generatedQueryExtension
```

[10] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context ClassTemplateInstantiation inv:
  self.generatedQueryExtension->forAll(n |
    n.child.name = self.queryRenamingExpression->
  select(r | r.namedQuery = n.parent).eval(self)->asSet)
```

Figure 11-6 on page 125 shows the abstract syntax of the templates package that describes association templates. An AssociationTemplate is an Association and therefore can be extended as described in the AssociationExtension chapter. An AssociationTemplate owns a set of template parameters and a set of renaming expressions.

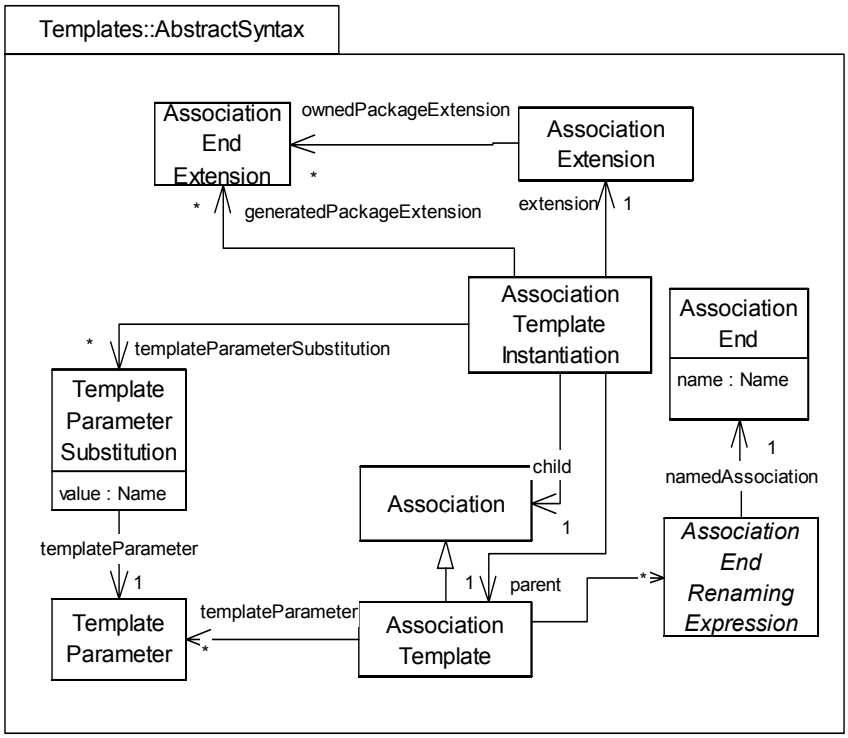


Figure 11-6 *Templates Abstract Syntax package (association templates)*

AssociationTemplate

A package template.

Associations

renamingExpression The renaming expressions that are associated with the namespace of the association template.

templateParameter The parameters of the association template.

AssociationTemplateInstantiation

An instantiation relationship between an association template and an association

Associations

child The association that results from the instantiation.

parent The association template.

templateParameterSubstitution The parameters that are substituted when instantiating the template.

generatedAssociationEndExtension The association end extensions that are generated to realise the instantiation.

11.2.5 Well-formedness Rules

AssociationTemplate

[1] Only one renaming expression per association end in a template.

```
context AssociationTemplate inv:
  self.associationEndRenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.namedAssociationEnd <> r2.namedAssociationEnd)
```

[2] Only association ends in the template's namespace have renaming expressions associated with them.

```
context AssociationTemplate inv:
  self.memberAssociationEnd->
    includesAll(self.associationEndRenamingExpression.namedAssociationEnd)
```

AssociationTemplateInstantiaton

[1] Parameter substitutions parameters must match those owned by the template.

```
context AssociationTemplateInstantiation inv:
  self.templateParameterSubstitutions.templateParameter =
  self.ownedParameter->asBag
```

[2] Association end substitutions are generated for each of the renamed association end in the parents namespace.

```
context AssociationE=TemplateInstantiation inv:
  self.generatedAssociationEndExtension.parent =
  self.extension.parent.associationEndRenamingExpression.namedAssociationEnd
```

[3] Generated association eend xtensions shadow redefined owned association end extensions.

```
context AssociationTemplateInstantiation inv:
  self.extension.ownedAssociationEndExtension->select(e | e.isRedefined) =
  self.generatedAssociationEndExtension
```

[4] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context AssociationTemplateInstantiation inv:
  self.generatedAssociationEndExtension->forAll(n |
    n.child.name = self.associationEndRenamingExpression->
      select(r | r.namedAssociationEnd = n.parent).eval(self)->asSet)
      select(r | r.namedClass = n.parent).eval(self)->asSet)
```

11.3 SEMANTIC DOMAIN

No additional semantics.

11.4 SEMANTIC MAPPING

No additional semantics.

11.5 EXAMPLE SNAPSHOTS

The snapshot in figure 11-8 on page 127 illustrates the example in Figure 11-7 on page 127. Note that instantiating the package template results in a generated class extension between the parameterised class in the template package P and the class B in package Q.

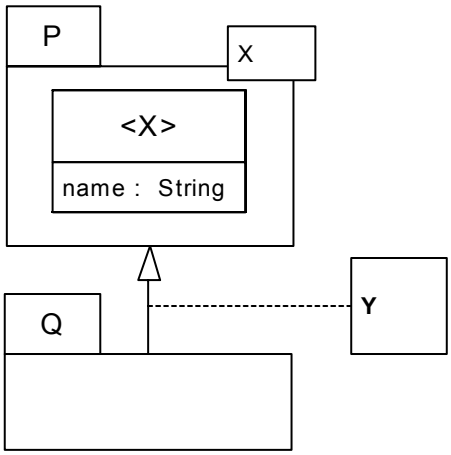


Figure 11-7 *Example template*

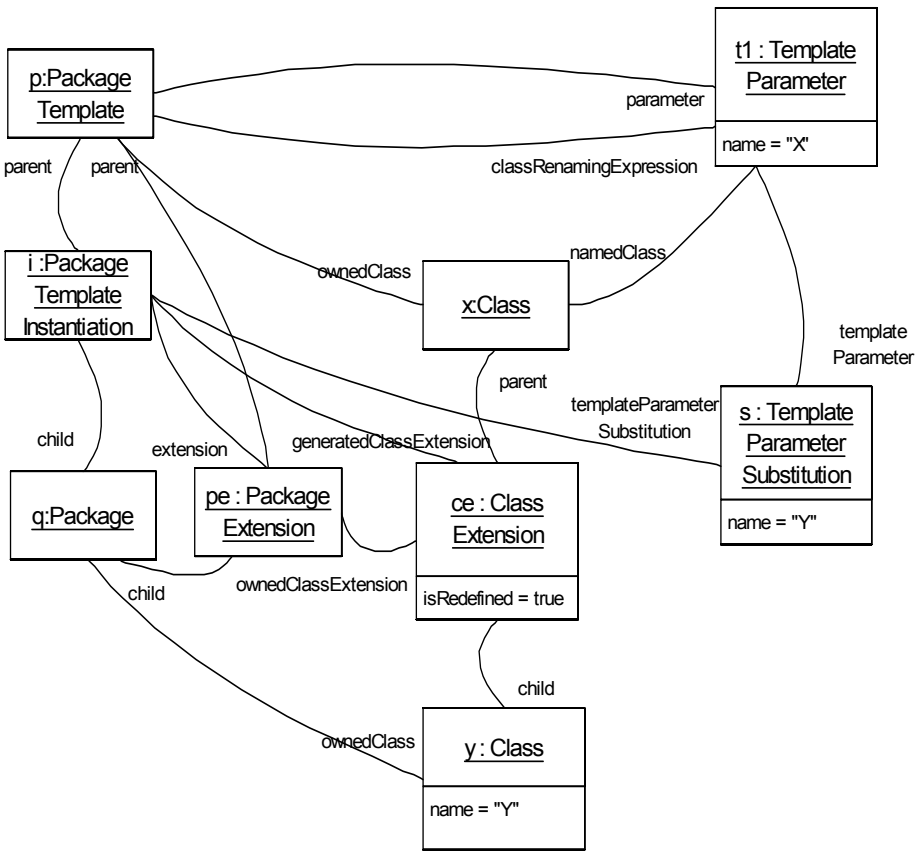


Figure 11-8 *Snapshot illustrating figure 11-7 on page 127*

11.6 CHANGES TO UML 1.4

UML 1.4 already provides support for templates but did not define their semantics.

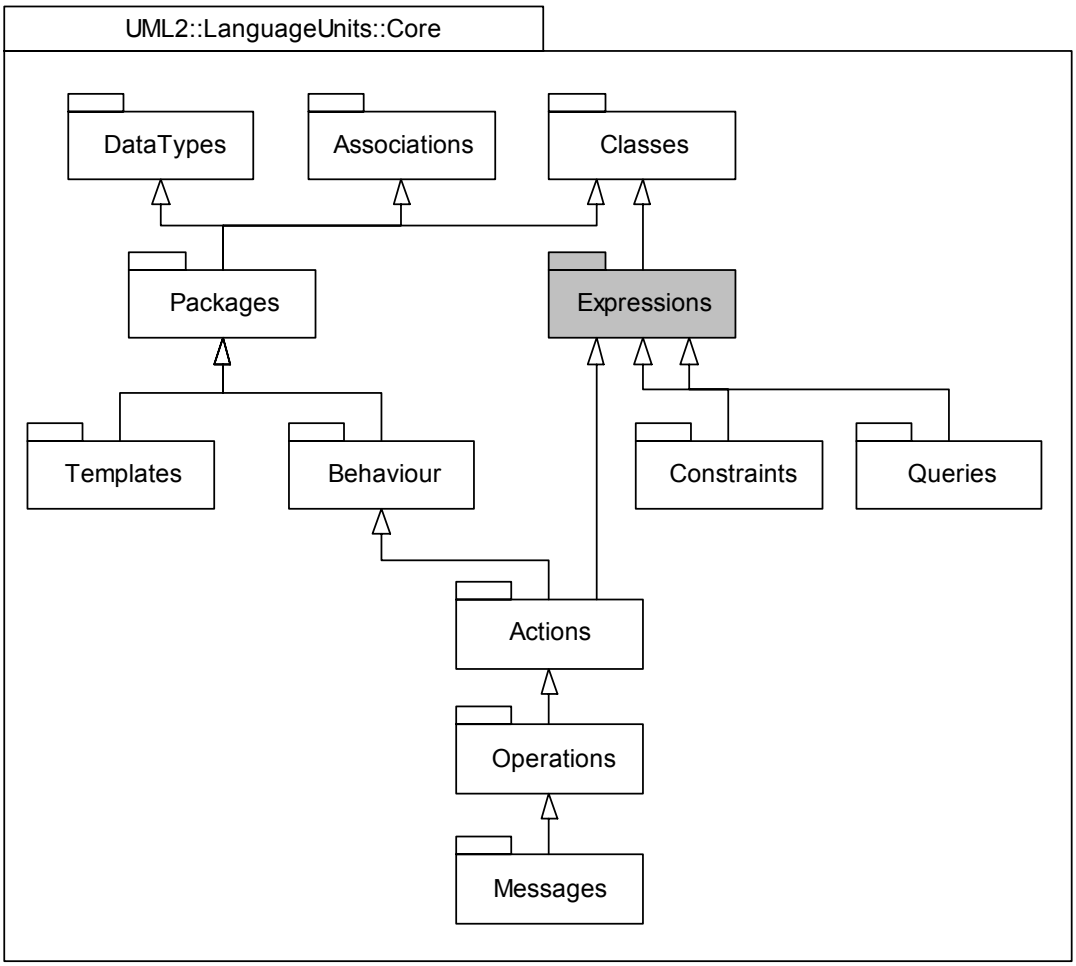
Chapter 12
Static Expressions

This package defines the abstract syntax and semantics of expressions. This chapter is mostly concerned with *static* expressions, which describe how computations take place which do not change the state of the system, and are used as a basis for describing constraints and queries (Chapters 13 and 14). Expressions that describe computations that do change the state of the system are called *actions* - these are covered in Chapter 16. The templates that are introduced towards the end of this chapter however are generic enough to be used for both static expressions and actions.

An expression has a return type, and its evaluations have values which must conform to that type. An expression may also have a number of operands, which are themselves expressions. The return type of an expression and its operand expressions may or may not need to be constrained, depending upon the actual expression. The operands can be thought of as sub-expressions of the originating expression. The operand expressions may have their own operands or sub-expressions, and in this way a hierarchy or expression tree may be formed. An expression also has a scope, which consists of one or more variable declarations - these declare the variables that may be referred to in any sub-expressions of the originating expression. The scope variable declarations are propagated down the expression hierarchy; ultimately bound variables at the leaves of the expression tree must point to a variable declaration that is within scope. Similarly an expression evaluation has an environment consisting of variable values, which provides the context for the evaluation, and a bound variable evaluation must similarly be within its environment.

This chapter presents the static expressions that lie at the core of the Object Constraint Language (OCL 2.0), an expression language incorporated into UML that is used to describe computations in object models. A complete definition of OCL is outside of the scope of this document - this can be found in the OCL 2.0 submission document [OCL 2.0]. The generic expression templates that allow a family of expression languages to be stamped out are introduced at the end of the chapter.

12.1 POSITION IN ARCHITECTURE



12.1.1 Example

A typical expression may look like the following:

```
bank.hasMoney and bank.hasStaff
```

This expression has two boolean sub-expressions "bank.hasMoney" and "bank.hasStaff", which are evaluated; the logical *and* operator is then applied to the two results, yielding an overall boolean value for the expression. This is a very simple expression, but shows that expressions can have sub-expressions, and when evaluated they yield a result of specified type.

This expression could either be used to form the basis of a constraint (that a bank must have both money and staff) or a query (a means of enquiring whether a bank has both money and staff). Thus every expression must have a context to show how its evaluation is used.

12.2 ABSTRACT SYNTAX

12.2.1 Derivation

Figure 12-1 on page 132 shows the derivation of the static expressions abstract syntax package using the abstract syntax templates described in sections 12.6.1 and 12.6.2.

Note that the type attribute inherited from expression is overridden for *not, and*, *equals*, *greater than* and *includes* expressions to be boolean.

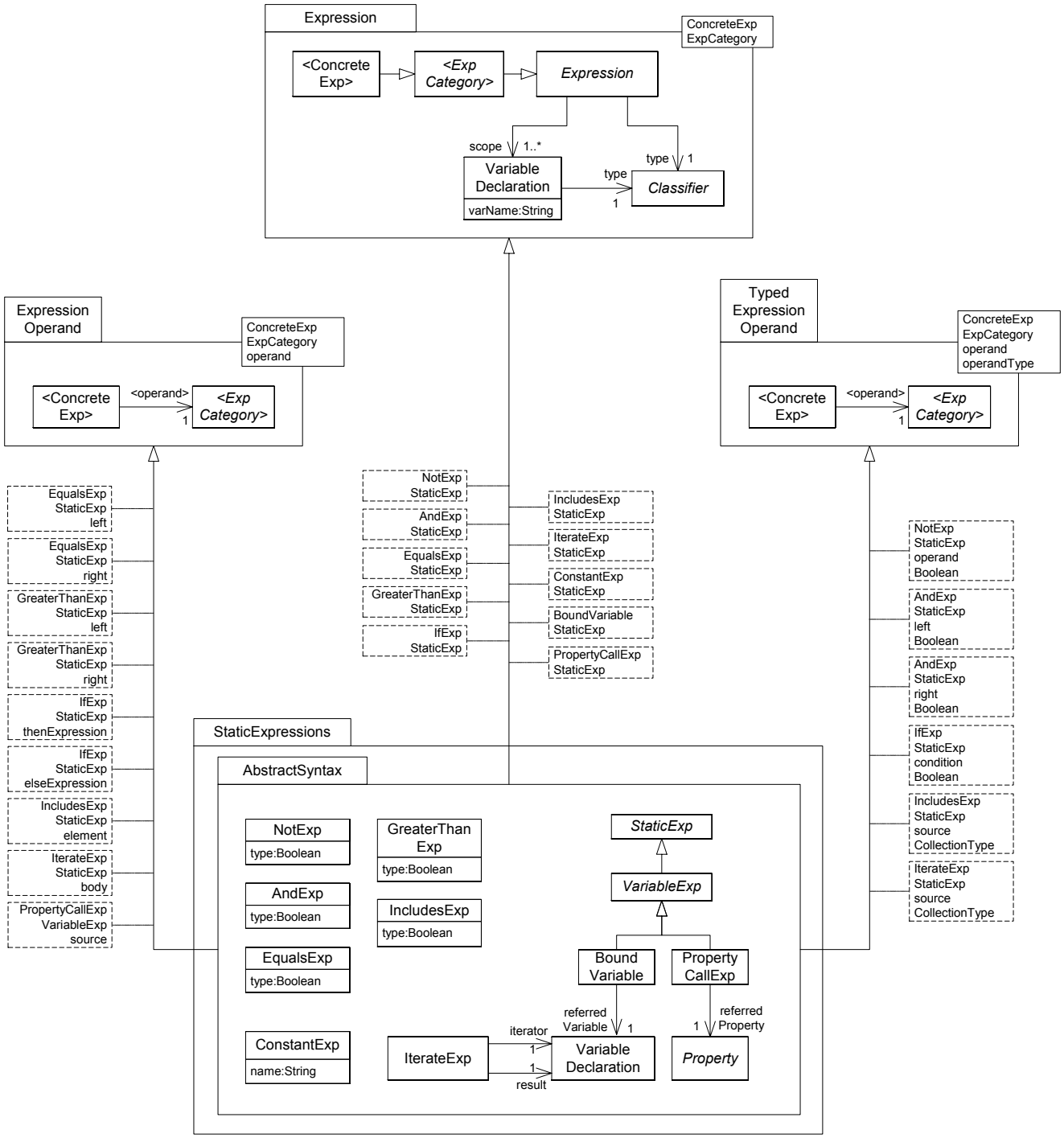


Figure 12-1 *Derivation of Static Expressions abstract syntax Package*

12.2.2 Model

Figure 12-2 on page 132 shows the stamped out abstract syntax of the static expressions package. A static expression is an expression whose evaluation does not change the state of the system. An expression can either be a static expression or an action (an expression whose evaluation changes the state of the system). An expression has a type that its evaluation must conform to, and a scope which consists of one or more variable declarations - these are variables that may be referred to in any sub-expressions (operand expressions) of that expression. Variable expressions are static expressions that contain a bound variable that is a reference to a variable declaration that has been introduced to its scope by another expression higher in the expression tree. Property call expressions return the value of a property (*e.g.* an attribute, query or association end) in relation to a particular source variable. An *iterate* expression evaluates a sub-expression for each element in a collection, and returns a value dependent upon that computation. An *if* expression returns one of two alternative values dependent upon the evaluation of a condition expression. A constant expression is a named expression that evaluates to an immutable value. *Not*, *and*, *equals*, *greater than* and *includes* expressions all return boolean values dependent upon the values of their operands.

Many of the descriptions of the modelling constructs in this and subsequent sections in this chapter are based upon those in the OCL 2.0 submission [OCL 2.0].

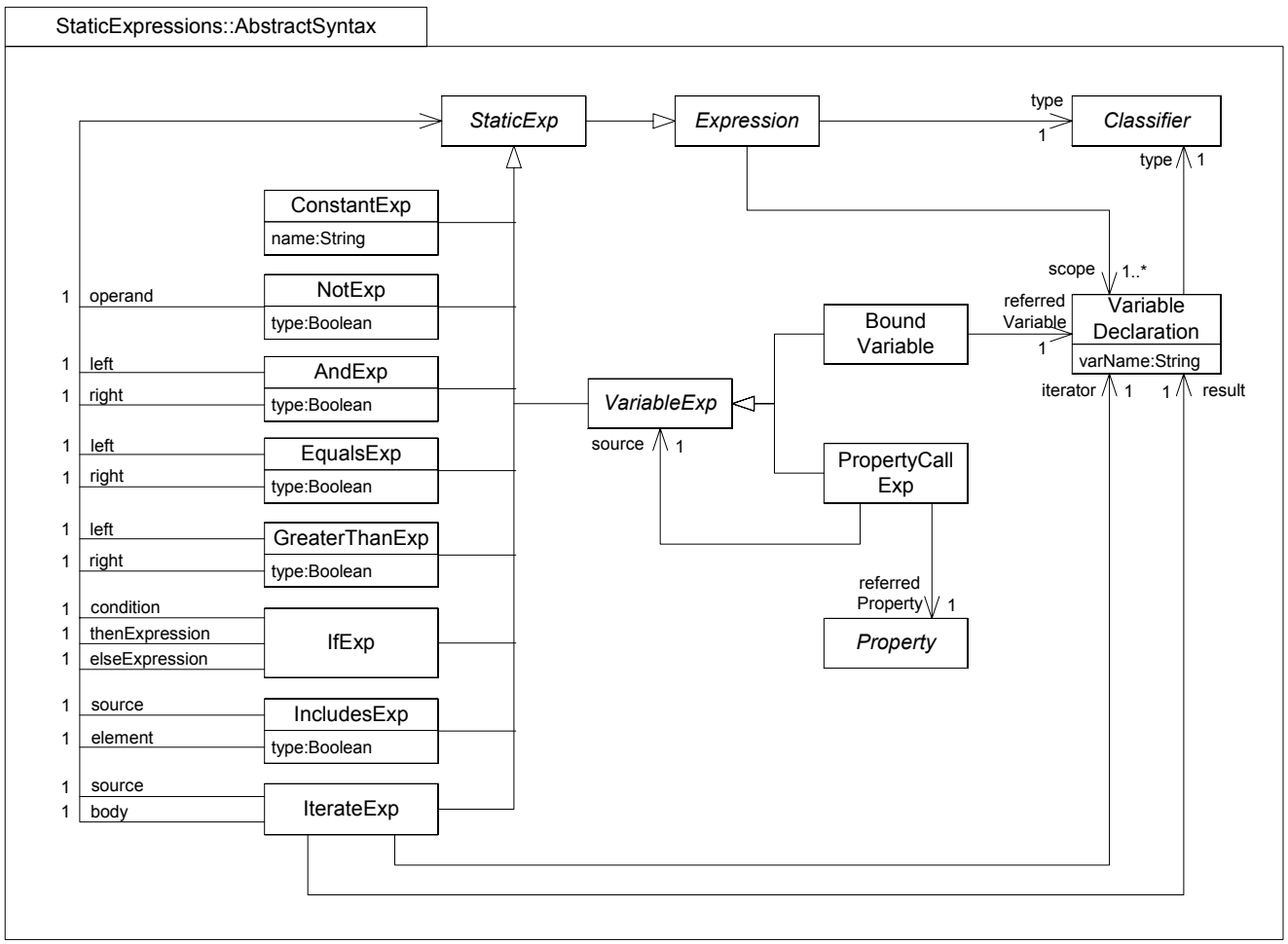


Figure 12-2 *Abstract syntax for Static Expressions package*

AndExp

An *and* expression is an expression that evaluates to the logical *and* of its two operand values.

Associations

left The left hand operand.

right The right hand operand.

type The return type of the expression (boolean).

BoundVariable

A bound variable is an expression that is a reference to variable declaration that is within scope (*i.e.* a variable that has been declared by another expression higher in the expression tree). Every expression has a variable "self" within its scope, which points to the object that owns the feature (such as a constraint or query) that provides the context for the evaluation.

Associations

referredVariable The variable declaration that the bound variable acts as a pointer to. This variable declaration must be within the scope of the bound variable.

ConstantExp

A constant is an expression that has a name and whose evaluation points to an immutable value.

Attributes

name The name of the constant.

EqualsExp

An *equals* expression is an expression that evaluates to the logical result of the equality test of its two operand values.

Associations

left The left hand operand.

right The right hand operand.

type The return type of the expression (boolean).

Expression

Expression is the abstract superclass for all expressions including static expressions and actions (see Chapter 16). An expression has a type which its evaluation must conform to, and a scope (one or more variable declarations that may be referred to in any operand sub-expressions.

Associations

type The return type of the expression.

scope The set of variable declarations that may be referred to within any operand sub-expressions.

GreaterThanExp

A *greater than* expression is an expression that evaluates to the logical result of testing whether its *left* operand value is greater than its *right* operand value.

Associations

left The left hand operand.

right The right hand operand.

type The return type of the expression (boolean).

IfExp

An *if* expression evaluates to the value of one of two alternative expressions, depending on the evaluation of the condition expression. Both the *then* and the *else* expressions are mandatory since the *if* expression must guarantee to result in a value.

Associations

condition The logical expression whose evaluation determines whether the value of the *then e*xpression (if the condition evaluates to true) or the *else* expression (if the condition evaluates to false) gets returned as the value of the *if* expression.

thenExpression The expression whose value is returned by the *if* expression if the condition expression evaluates to true.

elseExpression The expression whose value is returned by the *if* expression if the condition expression evaluates to false.

IncludesExp

An *includes* expression is an expression that evaluates to the logical result of testing whether its element operand value is a member of the collection returned by the source operand.

Associations

source The expression that returns a collection that the value of element is tested against.

element The expression whose value is tested to be within the collection returned by the source expression.

type The return type of the expression (boolean).

IterateExp

An *iterate* expression is an expression which evaluates its body expression for each element in the collection returned by the source expression, and returns a result whose value depends upon the computation.

Associations

source An expression that returns a collection - the body expression is then evaluated for each element in that collection.

body The expression that is evaluated for each member of the collection returned by the source expression.

iterator The variable that is bound to each element in the source collection whilst evaluating the body expression.

result The variable that represents the result returned by the evaluation of the *iterate* expression.

NotExp

A *not* expression is an expression that evaluates to the logical *not* of its operand value.

Associations

operand The boolean operand expression.

type The return type of the expression (boolean).

PropertyCallExp

A property call expression is an expression that refers to a property (*e.g.* an attribute, query or association end) of a particular source element, and which evaluates to the value of that property.

Associations

source The expression includes some bound variable whose value is used as the context for the referred property.

referredProperty The property whose value is returned by the property call expression. Properties include attributes, queries and association ends.

StaticExp

A static expression is an expression that does not change the state of the system (in contrast with an action). All the expressions described in this chapter are static expressions. Any static expression may be used to form the basis of a query (see Chapter 14) providing its type matches the query type, and any static expression that returns a boolean value may form the basis of a constraint (see Chapter 13) or an operation pre-condition or post-condition (see Chapter 17).

VariableDeclaration

A variable declaration binds a name to a type. Certain expressions, notably *iterate* expressions, introduce variable declarations which can be referred to in expressions where the variable is in scope (*i.e.* expressions lower down in the expression tree). In addition, every expression has a variable "self" within its scope, which points to the object whose class ultimately owns the expression - this is introduced by the context of the root expression in an expression tree (*e.g.* a constraint, query or operation). It is important to note that a variable declaration is not itself an expression.

Attributes

varName The name of the variable.

Associations

type The type of the variable.

VariableExp

A variable expression is an expression that contains a bound variable. A variable expression may be a property call expression or a bound variable itself.

12.2.3 Well-formedness rules

AndExp

[1] The scope of the left hand operand of an *and* expression must include all the variable declarations within the scope of the *and* expression.

```
context AndExp inv:
  self.left.scope -> includesAll(self.scope)
```

[2] The scope of the right hand operand of an *and* expression must include all the variable declarations within the scope of the *and* expression.

```
context AndExp inv:
  self.right.scope -> includesAll(self.scope)
```

[3] The left hand operand of an *and* expression must have a boolean return type.

```
context AndExp inv:
  self.left.type.isKindOf(Boolean)
```

[4] The right hand operand of an *and* expression must have a boolean return type.

```
context AndExp inv:
  self.right.type.isKindOf(Boolean)
```

BoundVariable

[1] The referred variable declaration of a bound variable must be within scope.

```
context BoundVariable inv:
  self.scope -> includes(self.referredVariable)
```

[2] The return type of a bound variable must match the type of the referred variable declaration.

```
context BoundVariable inv:
  self.type = self.referredVariable.type
```

EqualsExp

[1] The scope of the left hand operand of an *equals* expression must include all the variable declarations within the scope of the *equals* expression.

```
context EqualsExp inv:
  self.left.scope -> includesAll(self.scope)
```

[2] The scope of the right hand operand of an *equals* expression must include all the variable declarations within the scope of the *equals* expression.

```
context EqualsExp inv:
  self.right.scope -> includesAll(self.scope)
```

[3] The left and right hand operands of an *equals* expression must match.

```
context EqualsExp inv:
  self.left.type = self.right.type
```

Expression

[1] An expression cannot have two variable declarations with the same name within its scope.

```
context Expression inv:
  self.scope -> forAll(v1 |
     self.scope -> forAll(v2 |
        v1 <> v2 implies v1.varName <> v2.varName))
```

GreaterThanExp

[1] The scope of the left hand operand of a *greater than* expression must include all the variable declarations within the scope of the *greater than* expression.

```
context GreaterThanExp inv:
  self.left.scope -> includesAll(self.scope)
```

[2] The scope of the right hand operand of a *greater than* expression must include all the variable declarations within the scope of the *greater than* expression.

```
context GreaterThanExp inv:
  self.right.scope -> includesAll(self.scope)
```

[3] The left and right hand operands of a *greater than* expression must match.

```
context GreaterThanExp inv:
  self.left.type = self.right.type
```

IfExp

[1] The scope of the condition expression of an *if* expression must include all the variable declarations within the scope of the *if* expression.

```
context IfExp inv:
  self.condition.scope -> includesAll(self.scope)
```

[2] The scope of the *then* expression of an *if* expression must include all the variable declarations within the scope of the *if* expression.

```
context IfExp inv:
  self.thenExpression.scope -> includesAll(self.scope)
```

[3] The scope of the *else* expression of an *if* expression must include all the variable declarations within the scope of the *if* expression.

```
context IfExp inv:
  self.elseExpression.scope -> includesAll(self.scope)
```

[4] The condition expression of an *if* expression must have a boolean return type.

```
context IfExp inv:
  self.condition.type.isKindOf(Boolean)
```

[5] The return type of an *if* expression must match the types of both the *then* and *else* expressions.

```
context IfExp inv:
  self.type = self.thenExpression.type and
    self.type = self.elseExpression.type
```

IncludesExp

[1] The scope of the source expression of an *includes* expression must include all the variable declarations within the scope of the *includes* expression.

```
context IncludesExp inv:
  self.source.scope -> includesAll(self.scope)
```

[2] The scope of the element expression of an *includes* expression must include all the variable declarations within the scope of the *includes* expression.

```
context IncludesExp inv:
  self.element.scope -> includesAll(self.scope)
```

[3] The type of the source expression of an *includes* expression must be a collection type.

```
context IncludesExp inv:
  self.source.type.isKindOf(CollectionType)
```

[4] The type of the element expression in an *includes* expression must match the element type of the source collection.

```
context IncludesExp inv:
  self.element.type.isKindOf(self.source.type.elementType)
```

IterateExp

[1] The scope of the source expression of an *iterate* expression must include all the variable declarations within the scope of the *iterate* expression.

```
context IterateExp inv:
  self.source.scope -> includesAll(self.scope)
```

[2] The scope of the body expression of an *iterate* expression must include all the variable declarations within the scope of the *iterate* expression.

```
context IterateExp inv:
  self.body.scope -> includesAll(self.scope)
```

[3] The scope of the body expression of an *iterate* expression must include the iterator and result variable declarations.

```
context IterateExp inv:
  self.body.scope -> includes(self.iterator) and
  self.body.scope -> includes(self.result)
```

[4] The type of the source expression of an *iterate* expression must be a collection type.

```
context IterateExp inv:
  self.source.type.isKindOf(CollectionType)
```

[5] The type of the iterator variable in an *iterate* expression must match the element type of the source collection.

```
context IterateExp inv:
  self.iterator.type.isKindOf(self.source.type.elementType)
```

[6] The return type of an *iterate* expression must match the type of the result variable.

```
context IterateExp inv:
  self.type = self.result.type
```

NotExp

[1] The scope of the operand expression of a *not* expression must include all the variable declarations within the scope of the *not* expression.

```
context NotExp inv:
  self.operand.scope -> includesAll(self.scope)
```

[2] The operand expression of a *not* expression must have a boolean return type.

```
context NotExp inv:
  self.operand.type.isKindOf(Boolean)
```

PropertyCallExp

[1] The referred property of a property call expression must be one of the member properties of the return type of the source expression.

```
context PropertyCallExp inv:
  self.source.type.memberProperty -> includes(self.referredProperty)
```

[2] The return type of a property call expression must match the type of the referred property.

```
context PropertyCallExp inv:
  self.type = self.referredProperty.type
```

12.3 SEMANTIC DOMAIN

12.3.1 Derivation

Fig 12-3 on page 139 shows the derivation of the static expressions semantic domain package using the semantic domain templates described in sections 12.6.1 and 12.6.2.

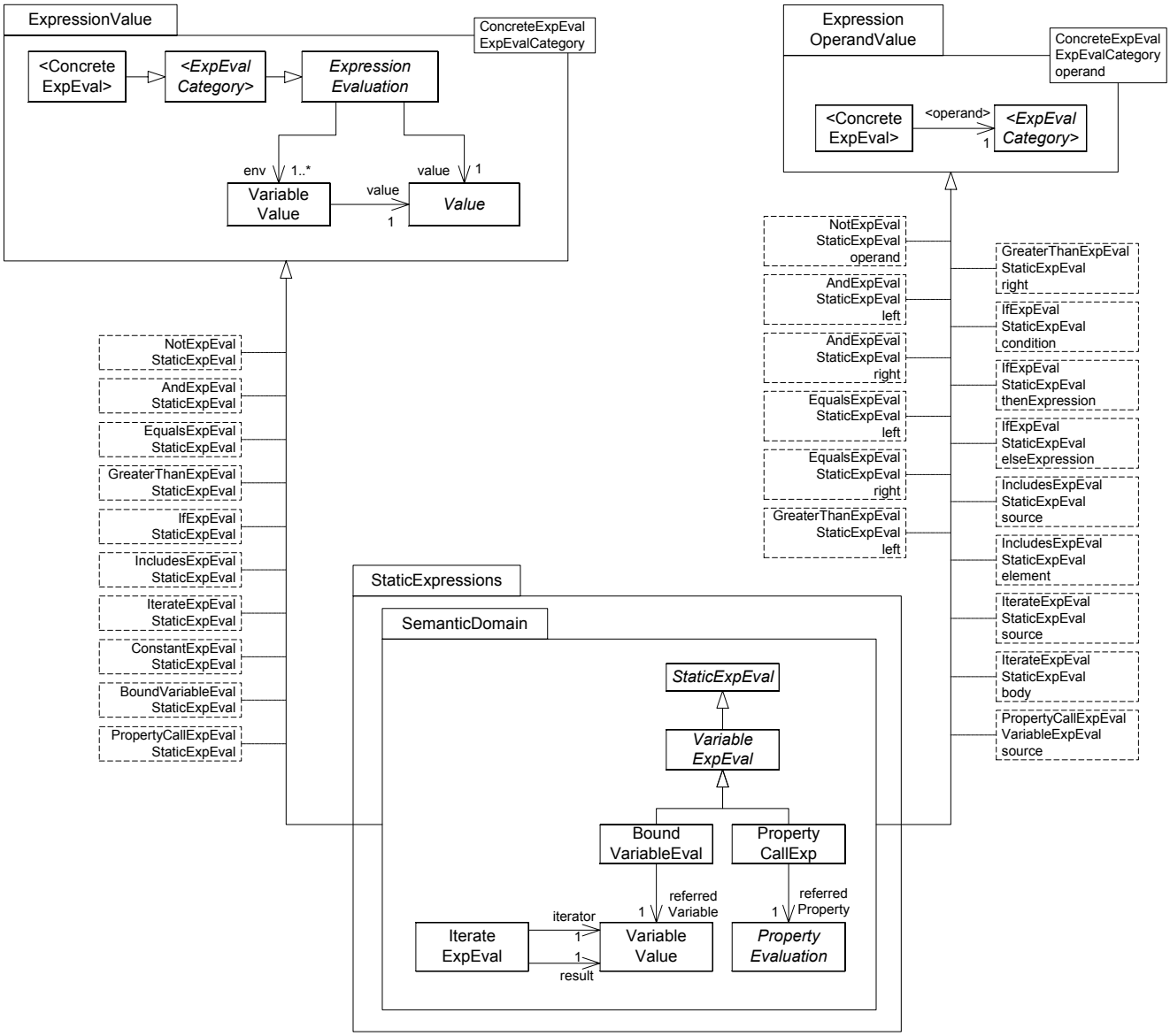


Figure 12-3 *Derivation of Static Expressions semantic domain Package*

12.3.2 Model

Fig. 12-4 on page 140 shows the stamped out semantic domain of the static expressions package. It defines the concepts that are necessary to express the meaning of static expressions. A static expression evaluation is one that does not change the state of the system (as opposed to an action evaluation).

An expression evaluation is an instance of an expression, and has a value and an environment, which consists of one or more variable values that may be used as the context of any operand sub-expression evaluations. A var-

iable value points to some value which is used as the basis for any variable expression evaluation. Property call expression evaluations return a property evaluation (which may be a slot, query or link end evaluation) in relation to a particular source variable value. Each of the other concrete expressions described in section 12.2 have an equivalent concrete expression evaluation - fuller descriptions for these are given in section 12.2.2 than are given below.

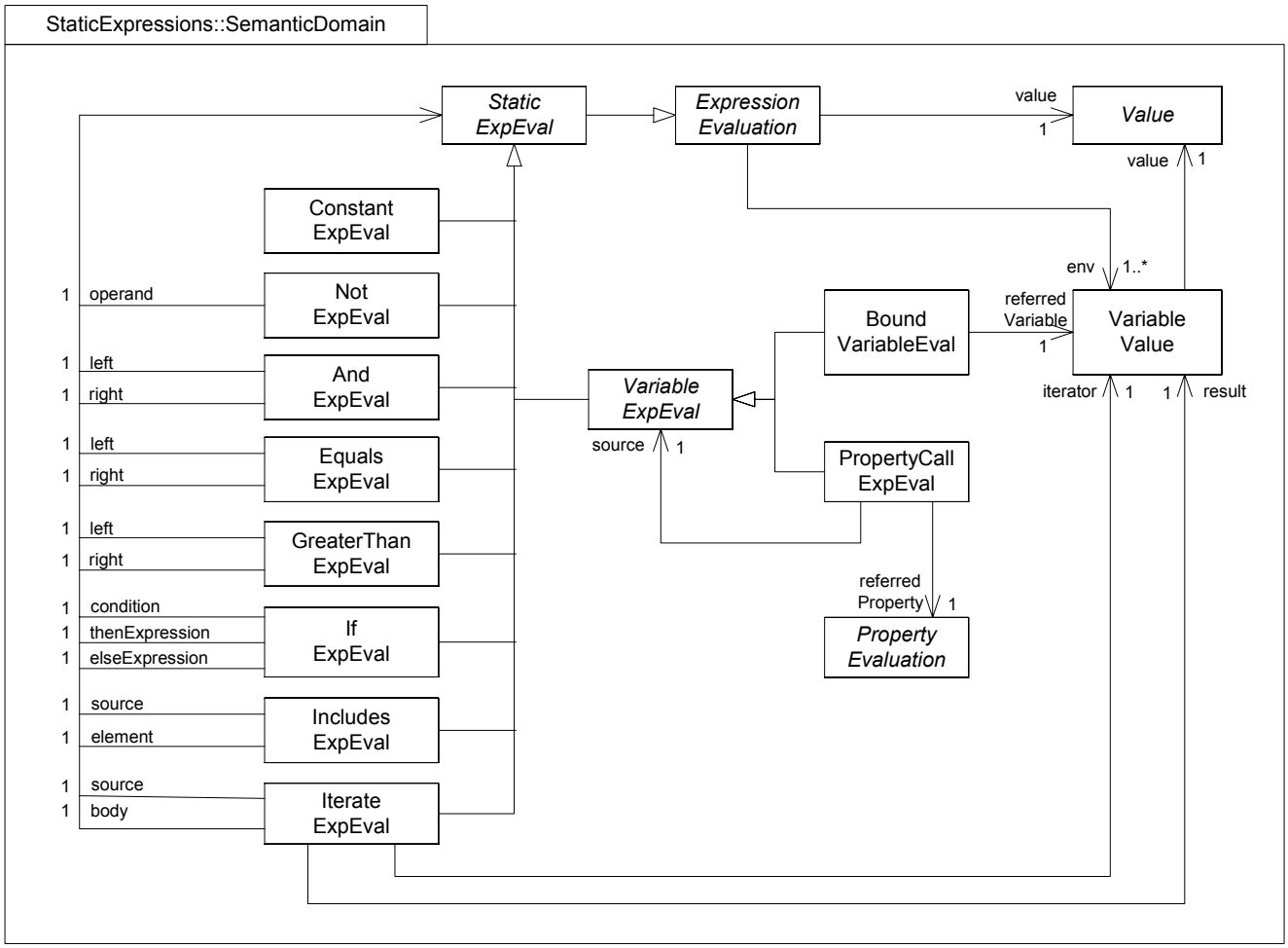


Figure 12-4 *Semantic domain for Static Expressions package*

AndExpEval

An *and* expression evaluation is an evaluation of an *and* expression.

Associations

left The evaluation of the left hand operand.

right The evaluation of the right hand operand.

BoundVariableEval

A bound variable evaluation is an evaluation of a bound variable. It points to a variable value which in turn points to a value that acts as the reference for a property call expression evaluation.

Associations

referredVariable Points to the variable value that in turns points to the reference value.

ConstantExpEval

A constant expression evaluation is an evaluation of a constant expression. It is a reference to some immutable value.

EqualsExpEval

An *equals* expression evaluation is an evaluation of an *equals* expression.

Associations

left The evaluation of the left hand operand.

right The evaluation of the right hand operand.

ExpressionEvaluation

Expression evaluation is the abstract superclass for all expression evaluations including static expression evaluation and action evaluations (see Chapter 16. An expression evaluation has a value and an environment, which consists of one or more variable values that may be used as the context of any operand sub-expression evaluations.

Associations

value The value of the expression evaluation.

env The set of variable values that form the environment.

GreaterThanExpEval

A *greater than* expression evaluation is an evaluation of a *greater than* expression.

Associations

left The evaluation of the left hand operand.

right The evaluation of the right hand operand.

IfExpEval

An *if* expression evaluation is an evaluation of in *if* expression.

Associations

condition The logical expression evaluation that determines whether the value of the *then* expression (if the condition is true) or the *else* expression (if the condition is false) gets returned as the value of the *if* expression evaluation.

thenExpression The expression evaluation that is returned by the *if* expression evaluation if the condition is true.

elseExpression The expression evaluation that is returned by the *if* expression evaluation if the condition is false.

IncludesExpEval

An *includes* expression evaluation is an evaluation of an *includes* expression.

Associations

source The expression evaluation that returns a collection that the element is tested against.

element The expression evaluation that is tested to be within the collection returned by the source.

IterateExpEval

An *iterate* expression evaluation is an evaluation of an *iterate* expression which evaluates its body expression for each element in the collection returned by the source expression, and returns a result whose value depends upon the computation.

Associations

source An expression evaluation that returns a collection - there is a body expression evaluation and iterator variable value for each element in that collection.

body The expression evaluations that are associated with each member of the collection returned by the *source*.

iterator The variable values that are bound to each element in the source collection and which are used in the body expression evaluations.

result The variable value that represents the result of the *iterate* expression evaluation.

NotExpEval

A *not* expression evaluation is an evaluation of a *not* expression.

Associations

operand The evaluation of the operand expression.

PropertyCallExpEval

A property call expression evaluation is an evaluation of a property call expression. It refers to a property evaluation (*e.g.* a slot, query or link end evaluation).

Associations

source The expression evaluation that includes some bound variable value that is used as the context for the referred property.

referredProperty The property evaluation that is returned by the property call expression. Property evaluations include slot values, query evaluations and link end evaluations.

StaticExpEval

A static expression evaluation is an expression evaluation that does not change the state of the system. All expression evaluations described in this chapter are static expression evaluations. Static expression evaluations may form the basis of evaluations of queries, constraints and operation pre-conditions and post-conditions.

VariableExpEval

A variable expression evaluation is an evaluation of a variable expression. A variable expression evaluation may be a property call expression evaluation or a bound variable evaluation.

VariableValue

A variable value is an instance of a variable declaration, and is a reference to some value which provides the context for property call expression evaluations. It is important to note that a variable value is not an expression evaluation.

Associations

value The value of the variable.

12.3.3 Well-formedness rules

AndExpEval

[1] The environment of the left hand operand of an *and* expression evaluation must include all the variable values within the environment of the *and* expression evaluation.

```
context AndExpEval inv:
  self.left.env -> includesAll(self.env)
```

[2] The environment of the right hand operand of an *and* expression evaluation must include all the variable values within the environment of the *and* expression evaluation.

```
context AndExpEval inv:
  self.right.env -> includesAll(self.env)
```

BoundVariableEval

[1] The referred variable value of a bound variable evaluation must be within that evaluation's environment.

```
context BoundVariableEval inv:
  self.env -> includes(self.referredVariable)
```

[2] The value of a bound variable evaluation must be the same as its referred variable's value.

```
context BoundVariableEval inv:
  self.value = self.referredVariable.value
```

EqualsExpEval

[1] The environment of the left hand operand of an *equals* expression evaluation must include all the variable values within the environment of the *equals* expression evaluation.

```
context EqualsExpEval inv:
  self.left.env -> includesAll(self.env)
```

[2] The environment of the right hand operand of an *equals* expression evaluation must include all the variable values within the environment of the *equals* expression evaluation.

```
context EqualsExpEval inv:
  self.right.env -> includesAll(self.env)
```

GreaterThanExpEval

[1] The environment of the left hand operand of a *greater than* expression evaluation must include all the variable values within the environment of the *greater than* expression evaluation.

```
context GreaterThanExpEval inv:
  self.left.env -> includesAll(self.env)
```

[2] The environment of the right hand operand of a *greater than* expression evaluation must include all the variable values within the environment of the *greater than* expression evaluation.

```
context GreaterThanExpEval inv:
  self.right.env -> includesAll(self.env)
```

IfExpEval

[1] The environment of the condition expression evaluation of an *if* expression evaluation must include all the variable values within the environment of the *if* expression evaluation.

```
context IfExpEval inv:
  self.condition.env -> includesAll(self.env)
```

[2] The environment of the *then* expression evaluation of an *if* expression evaluation must include all the variable values within the environment of the *if* expression evaluation.

```
context IfExpEval inv:
  self.thenExpression.env -> includesAll(self.env)
```

[3] The environment of the *else* expression evaluation of an *if* expression evaluation must include all the variable values within the environment of the *if* expression evaluation.

```
context IfExpEval inv:
  self.elseExpression.env -> includesAll(self.env)
```

IncludesExpEval

[1] The environment of the source expression evaluation of an *includes* expression evaluation must include all the variable values within the environment of the *includes* expression evaluation.

```
context IncludesExpEval inv:
  self.source.env -> includesAll(self.env)
```

[2] The environment of the element expression evaluation of an *includes* expression evaluation must include all the variable values within the environment of the *includes* expression evaluation.

```
context IncludesExpEval inv:
  self.element.env -> includesAll(self.env)
```

IterateExpEval

[1] The environment of the source expression evaluation of an *iterate* expression evaluation must include all the variable values within the environment of the *iterate* expression evaluation.

```
context IterateExpEval inv:
  self.source.env -> includesAll(self.env)
```

[2] The environment of the body expression evaluation of an *iterate* expression evaluation must include all the variable values within the environment of the *iterate* expression evaluation.

```
context IterateExpEval inv:
  self.body.env -> includesAll(self.env)
```

[3] The environment of the body expression evaluation of an *iterate* expression evaluation must include the iterator and result variable values.

```
context IterateExpEval inv:
  self.body.env -> includes(self.iterator) and
  self.body.env -> includes(self.result)
```

NotExpEval

[1] The environment of the operand of a *not* expression evaluation must include all the variable values within the environment of the *not* expression evaluation.

```
context NotExpEval inv:
```

```
self.operand.env -> includesAll(self.env)
```

PropertyCallExpEval

[1] The environment of the source expression evaluation of an property call expression evaluation must include all the variable values within the environment of the property call expression evaluation.

```
context PropertyCallExpEval inv:
  self.source.env -> includesAll(self.env)
```

[2] The referred property evaluation of a property call expression evaluation must be one of the owned property evaluations of the value of the source expression evaluation.

```
context PropertyCallExpEval inv:
  self.source.value.ownedPropertyEval -> includes(self.referredProperty)
```

[3] The value of a property call expression evaluation must be the same as its referred property's value.

```
context PropertyCallExpEval inv:
  self.value = self.referredProperty.value
```

12.4 SEMANTIC MAPPING

12.4.1 Derivation

Fig 12-5 on page 146 shows the derivation of the static expressions semantic domain package using the semantic mapping templates described in sections 12.6.1 and 12.6.2. These templates ensure that each expression evaluation in the semantic domain is mapped to the appropriate expression in the abstract syntax, and that operand evaluations are mapped to the corresponding operand expressions.

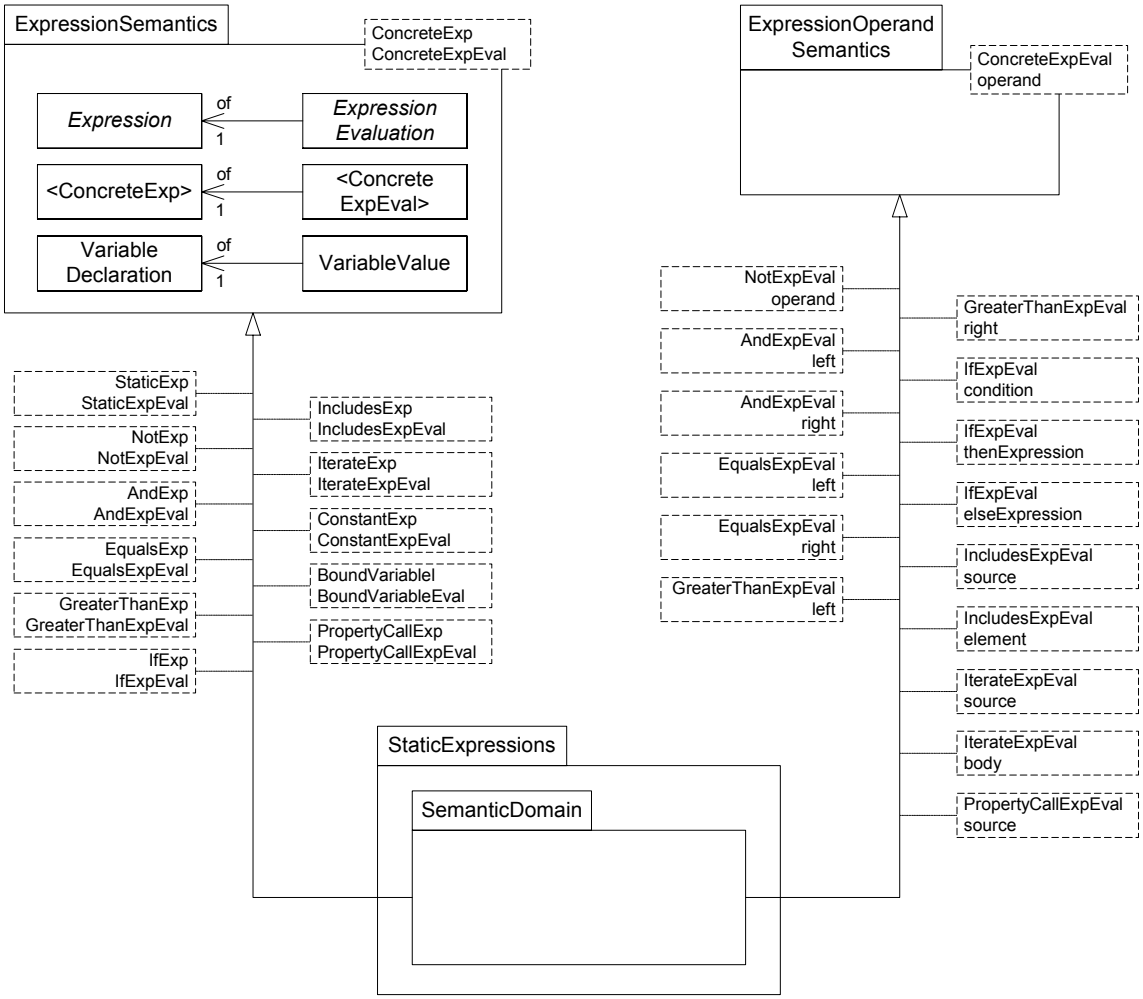


Figure 12-5 *Derivation of Static Expressions semantic mapping package*

12.4.2 Model

The semantic mappings package for expressions is shown in 12-6 on page 147. It defines the relationship that holds between expressions and their evaluations. An expression evaluation is an instance of an expression, and the meaning of an expression is defined by the set of all possible evaluations that can be assigned to the expression. In addition, a variable value is an instance of a variable declaration (which is not an expression). For an expression evaluation to be a valid instance of an expression, its value must conform to the type of that expression, and any operand values must also conform to the operand types in that expression.

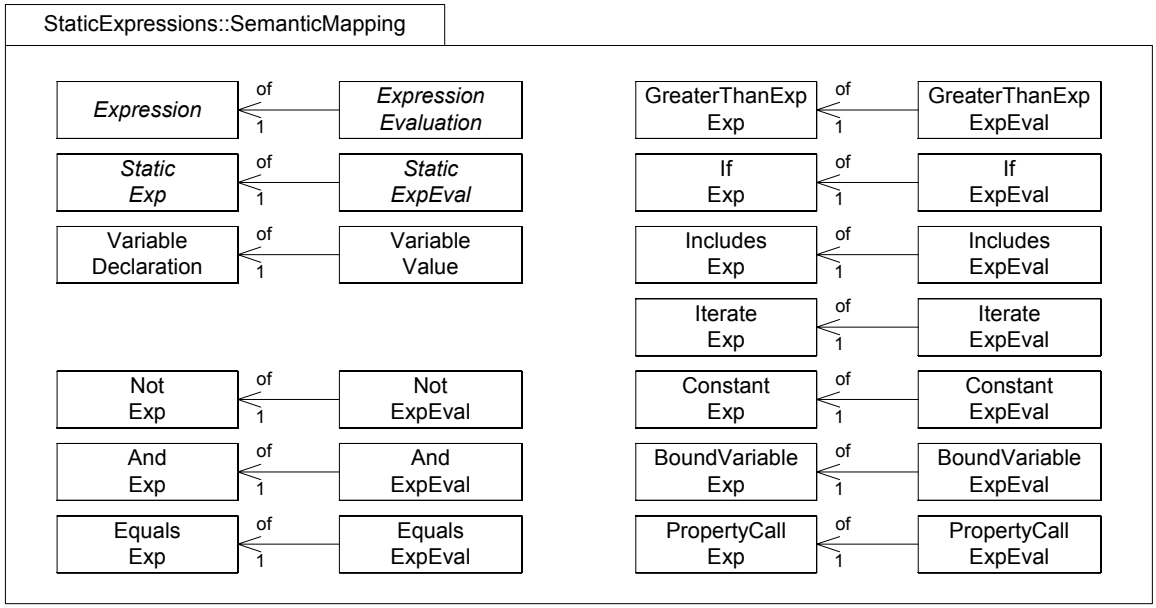


Figure 12-6 *Semantic Mapping for the Static Expressions Package*

12.4.3 Well-formedness rules

AndExpEval

[1] An *and* expression evaluation's left hand operand commutes with the corresponding expression's left hand operand.

```
context AndExpEval inv:
  self.left.of = self.of.left
```

[2] An *and* expression evaluation's right hand operand commutes with the corresponding expression's right hand operand.

```
context AndExpEval inv:
  self.right.of = self.of.right
```

BoundVariableEval

[1] A bound variable evaluation's referred variable value commutes with the corresponding bound variable's referred variable declaration.

```
context BoundVariableEval inv:
  self.referredVariable.of = self.of.referredVariable
```

EqualsExpEval

[1] An *equals* expression evaluation's left hand operand commutes with the corresponding expression's left hand operand.

```
context EqualsExpEval inv:
  self.left.of = self.of.left
```

[2] An *equals* expression evaluation's right hand operand commutes with the corresponding expression's right hand operand.

```
context EqualsExpEval inv:
  self.right.of = self.of.right
```

ExpressionEvaluation

[1] The value of an expression evaluation should conform to its expression's type.

```
context ExpressionEvaluation inv:
  self.value.of.conformsTo(self.of.type)
```

[2] An expression evaluation should have a variable value within its environment for every variable declaration within the scope of the corresponding expression.

```
context ExpressionEvaluation inv:
  self.of.scope -> forAll(v |
     self.env -> exists(vv | vv.of=v))
```

[3] For each variable value within the environment of an expression evaluation, there should be a variable declaration within the scope of the corresponding expression.

```
context ExpressionEvaluation inv:
  self.env -> forAll(vv |
     self.of.scope -> exists(v | vv.of=v))
```

GreaterThanExpEval

[1] A *greater than* expression evaluation's left hand operand commutes with the corresponding expression's left hand operand.

```
context GreaterThanExpEval inv:
  self.left.of = self.of.left
```

[2] A *greater than* expression evaluation's right hand operand commutes with the corresponding expression's right hand operand.

```
context GreaterThanExpEval inv:
  self.right.of = self.of.right
```

IfExpEval

[1] An *if* expression evaluation's condition operand commutes with the corresponding expression's condition operand.

```
context IfExpEval inv:
  self.condition.of = self.of.condition
```

[2] An *if* expression evaluation's *then* operand commutes with the corresponding expression's *then* operand.

```
context IfExpEval inv:
  self.thenExpression.of = self.of.thenExpression
```

[3] An *if* expression evaluation's *else* operand commutes with the corresponding expression's *else* operand.

```
context IfExpEval inv:
  self.elseExpression.of = self.of.elseExpression
```

IncludesExpEval

[1] An *includes* expression evaluation's source operand commutes with the corresponding expression's source operand.

```
context IncludesExpEval inv:
  self.source.of = self.of.source
```

[2] An *includes* expression evaluation's element operand commutes with the corresponding expression's element operand.

```
context IncludesExpEval inv:
  self.element.of = self.of.element
```

IterateExpEval

[1] An *iterate* expression evaluation's source operand commutes with the corresponding expression's source operand.

```
context IterateExpEval inv:
  self.source.of = self.of.source
```

[2] An *iterate* expression evaluation's body operand commutes with the corresponding expression's body operand.

```
context IterateExpEval inv:
  self.body.of = self.of.body
```

[3] An *iterate* expression evaluation's iterator variable value commutes with the corresponding expression's iterator variable declaration.

```
context IterateExpEval inv:
  self.iterator.of = self.of.iterator
```

[4] An *iterate* expression evaluation's result variable value commutes with the corresponding expression's result variable declaration.

```
context IterateExpEval inv:
  self.result.of = self.of.result
```

NotExpEval

[1] A *not* expression evaluation's operand commutes with the corresponding expression's operand.

```
context NotExpEval inv:
  self.operand.of = self.of.operand
```

PropertyCallExpEval

[1] A property call expression evaluation's source operand commutes with the corresponding expression's source operand.

```
context PropertyCallExpEval inv:
  self.source.of = self.of.source
```

[2] A property call expression evaluation's referred property evaluation commutes with the corresponding expression's referred property.

```
context PropertyCallExpEval inv:
  self.referredProperty.of = self.of.referredProperty
```

VariableValue

[1] The value of a variable value should conform to its variable declaration's type.

```
context VariableValue inv:
  self.value.of.conformsTo(self.of.type)
```

12.5 EXAMPLE SNAPSHOTS

Figure 12-8 on page 151 shows a partial snaphot of the constraint shown in figure 12-7 on page 150. This snapshot is concerned largely with showing the relationship between expressions (and their evaluations) in an expression tree, how the variables within scope and environment are propagated. As a result, the constraint itself is omitted for brevity - an alternative partial view of the same snapshot can be found in the constraints chapter (Chapter 13), where the relationship between a constraint and its body expression is depicted.

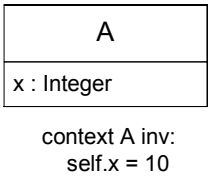


Figure 12-7 *Example class and constraint*

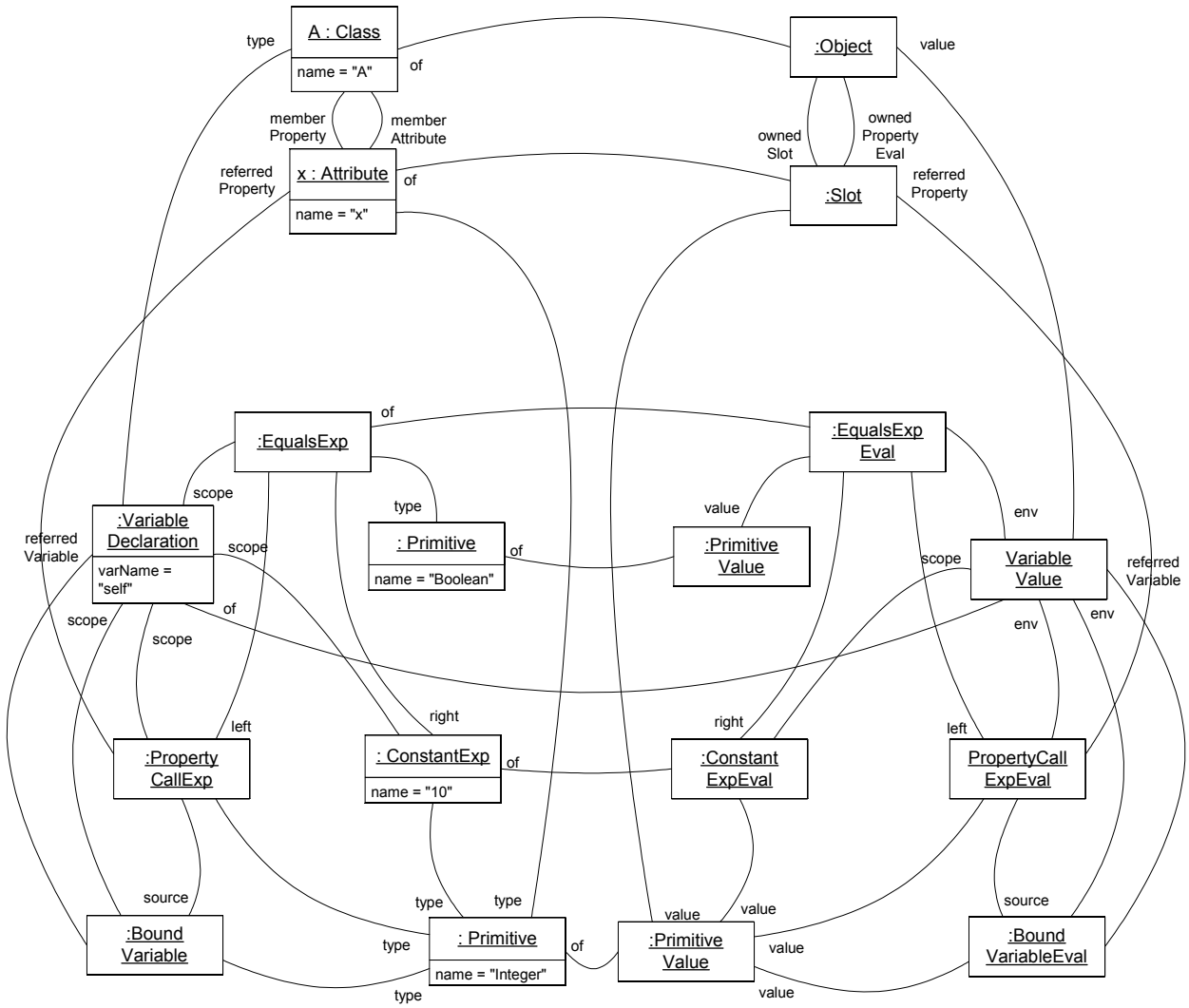


Figure 12-8 *Snapshot of Static Expression semantic mapping package*

12.6 TEMPLATES

This section introduces a set of generic templates which capture the essence of expressions, and can be used to stamp out a family of expression languages. Sections 12.2 to 12.4 then shows how these templates can be used to stamp out the core of OCL 2.0.

12.6.1 Expression

Expressions have a type and a scope (a set of variable declarations), and their evaluations have a value and an environment (a set of variable values), which provides the context for the evaluation.

Figure 12-9 on page 152 shows the abstract syntax for expressions. An expression has a type - this may be further constrained for a stamped out concrete expression (for example, an *and* expression has a boolean type). An expression also has a scope, which consists of one or more variable declarations - these declare the variables that may be referred to in any sub-expressions of the originating expression. A variable declaration also has a type,

and a name by which it is referred. Expressions are grouped into categories (static expressions and actions), and each concrete expression belongs to a particular category.

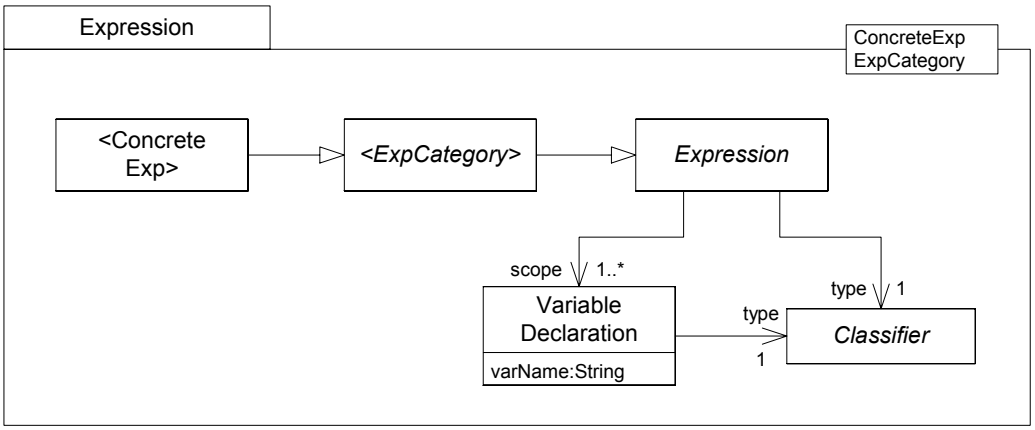


Figure 12-9 *Expression (abstract syntax) template*

An expression cannot have two variable declarations with the same name within its scope. This is expressed using the following constraint:

```
context Expression inv:
  self.scope -> forAll(v1 |
     self.scope -> forAll(v2 |
        v1 <> v2 implies v1.varName <> v2.varName))
```

Figure 12-10 on page 152 shows the semantic domain for expressions. An expression evaluation has a value (for example, an *and* expression evaluation must return a *Boolean* value), and is bound to a set of variable values, which represents the environment or context for the evaluation. A variable value is in effect a pointer to a value. Expression evaluations are grouped into categories (static expression evaluations and action evaluations), and each concrete expression evaluation belongs to a particular category.

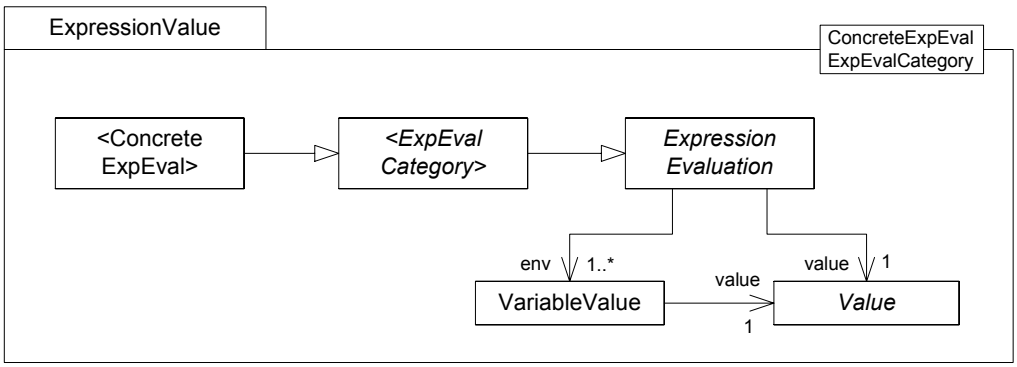


Figure 12-10 *Expression value (semantic domain) template*

Figure 12-11 on page 153 shows the semantic mapping for expressions, which associates expression evaluations with expressions, and variable values with variable declarations. The meaning of an expression is defined by the set of valid evaluations, and the meaning of a variable declaration is defined by the set of valid variable values. It should be noted that this template is stamped out from the basic semantics template, but its derivation is not explicitly shown here.

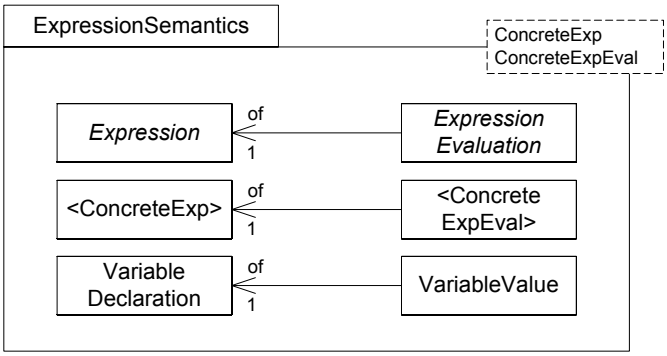


Figure 12-11 *Expression semantics template*

The value of an expression evaluation should be valid in view of its type:

```
context ExpressionEvaluation inv:
  self.value.of.conformsTo(self.of.type)
```

An expression evaluation should have a variable value within its environment for every variable declaration within the scope of its corresponding expression:

```
context ExpressionEvaluation inv:
  self.of.scope -> forAll(v |
     self.env -> exists(vv | vv.of=v))
```

For each variable value within the environment of an expression evaluation, there should be a variable declaration within the scope of its corresponding expression:

```
context ExpressionEvaluation inv:
  self.env -> forAll(vv |
     self.of.scope -> exists(v | vv.of=v))
```

The value of a variable value should conform to its variable declaration's type:

```
context VariableValue inv:
  self.value.of.conformsTo(self.of.type)
```

12.6.2 Expression operands

Expressions have operands upon which they act, which are themselves expressions. The type of those operand expressions must sometimes be constrained (for example the operands of a logical expression such as *and* or *not* must have a boolean return type). The variable declarations that are within the scope of an expression gets propagated down to its operand (sub-)expressions, and similarly for the variable values within the environment of an expression evaluation. In this section, templates are introduced that allow one or more operands to be added to expressions, along with corresponding semantic domain and semantic mapping templates. Each template adds a single operand - the templates can be stamped out multiple times for multiple operands.

Figure 12-12 on page 154 shows the two abstract syntax templates for expression operands. The upper template is a basic operand template, which adds to an expression a single operand, which is itself an expression. The lower template augments the first by adding a constraint on the return type of the operand.

It should be noted that semantic domain and semantic mapping templates for typed expression operands are not required, since expression values are already checked against type in the expression operand semantics template (see below).

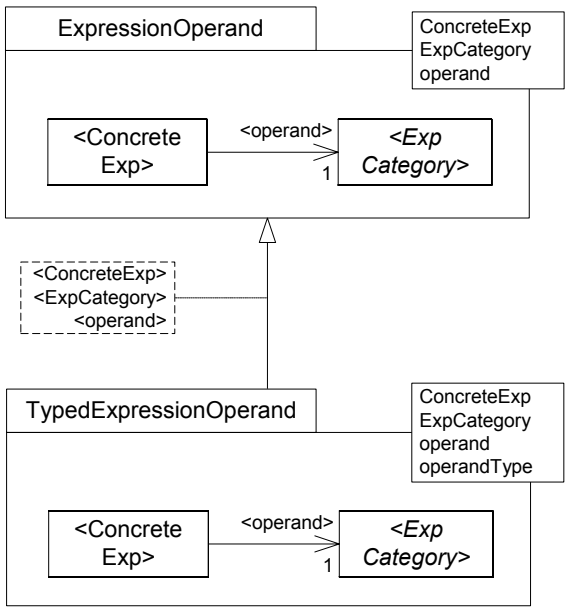


Figure 12-12 *Expression operand (abstract syntax) templates*

A crucial aspect of expressions is that their scope is propagated down to their operand sub-expressions; *i.e.* whatever variable declarations are within the scope of an expression are also within the scope of that expression's operand sub-expressions. This is expressed using the following constraint:

```
context <ConcreteExp> inv:
  self.<operand>.scope -> includesAll(self.scope)
```

In addition, within the typed expression operand template, an operand's type should match the type specified in the parameters:

```
context <ConcreteExp> inv:
  self.<operand>.type.isKindOf(<operandType>)
```

Figure 12-13 on page 154 shows the semantic domain template for expression operands. An expression evaluation has an operand, which is itself an expression evaluation.

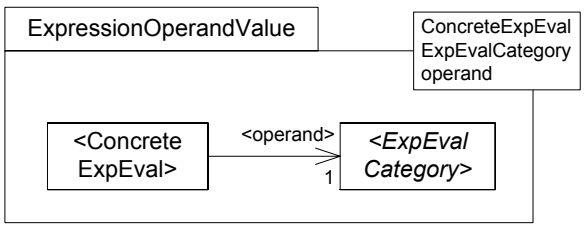


Figure 12-13 *Expression operand value (semantic domain) template*

As with expression scope, the environment of an expression evaluation is propagated down to its operand sub-expression evaluations:

```
context <ConcreteExpEval> inv:
  self.<operand>.env -> includesAll(self.env)
```

Figure 12-14 on page 155 shows the semantic mapping template for expression operands. The template contains no classes, as it simply adds a constraint to the model that would be stamped out from the abstract syntax and semantic domain templates.

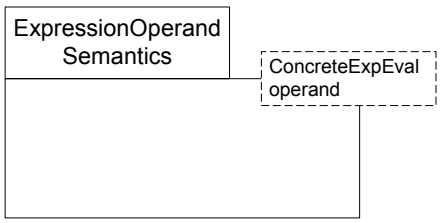


Figure 12-14 *Expression operand semantics template*

An expression evaluation's operands commute with the corresponding expression's operands:

```
context <ConcreteExpEval> inv:
  self.<operand>.of = self.of.<operand>
```

12.6.3 Expression context

Expressions cannot exist in isolation - they must always relate to some context, such as a class constraint or query, or an operation pre- or post-condition. It is the responsibility of the expression context to introduce one or more variable declarations (such as "self" or any parameters) to the scope of their root expression. These variable declarations are then propagated down the expression hierarchy as described in expression operands section (section 12.6.2). Similarly, instances of these expression context elements introduce corresponding variable values to the scope of their root expression evaluation. These templates introduce a single variable to the scope; for multiple variables, the templates can be stamped out more than once.

The templates in this section are not actually used to stamp out expressions themselves, and hence they are not used in this chapter. Instead they are used to stamp out any context for expressions such are constraints, queries and operations.

Figure 12-15 on page 155 shows the abstract syntax template for an expression context.

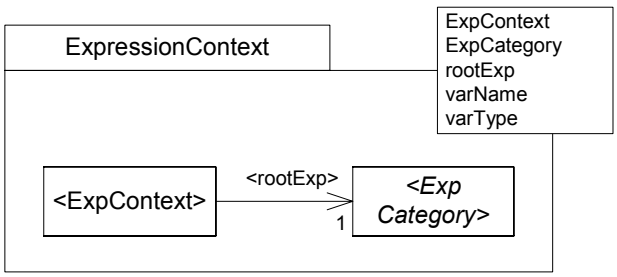


Figure 12-15 *Expression Context (Abstract Syntax) Template*

An expression context introduces one or more variable declarations into the scope of its root expression using the following constraint:

```
context <ExpContext> inv:
  self.<rootExp>.scope -> exists(v | v.varName=<varName> and v.type=<varType>)
```

Figure 12-16 on page 156 shows the semantic domain template for an expression context.

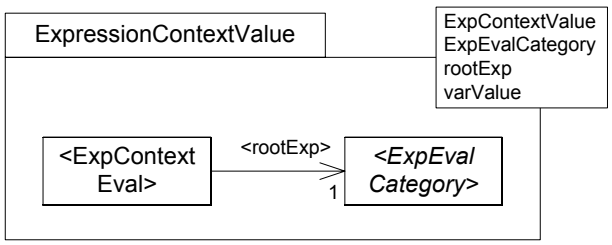


Figure 12-16 *Expression context (semantic domain) template*

An expression context evaluation introduces one or more variable values into the scope of its root expression evaluation:

```
context <ExpContextEval> inv:
  self.<rootExp>.scope -> exists(v | v.value=<varValue>)
```

No semantic mapping template is required for expression context, as variable values and variable declarations are already matched up via the expression semantic mapping constraints in section 12.6.1.

12.7 CHANGES FROM UML 1.4

UML 1.4 defines expressions as strings. This submission aims to provide a fuller definition that is compatible with the OCL 2.0 submission.

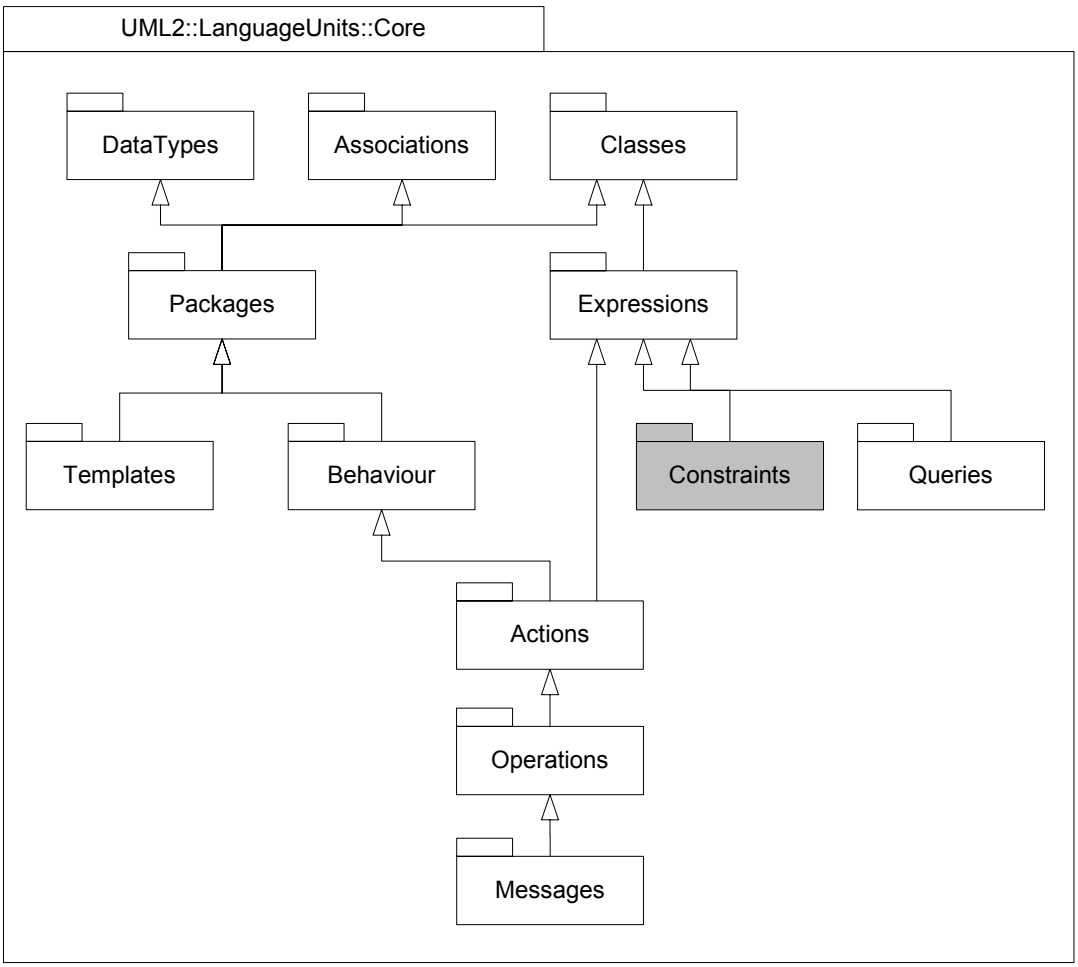
12.8 RELATIONSHIP TO OCL 2.0 SUBMISSION

- The goal of this submission has not been to match the inheritance hierarchy of the OCL 2.0 submission exactly (there is no loop expression for example), but the flattened OCL 2.0 model, as templates are used in place of abstract classes unless polymorphism is required.
- There are no separate property call expressions for individual properties (there is no attribute call expression for example) - instead the abstract property class is used as a plug-in point.
- There is only one generic iterate expression rather than the iterate expression and iterator expression for simplicity in the OCL 2.0 submission.
- Namespaces (a key part of the OCL 2.0 semantic domain) are not covered in this chapter as they are described in Chapter 7. Similarly action expressions are covered in Chapter 16.
- A variable declaration is not an expression, as this would mean it could be substituted anywhere an expression is expected - only certain expressions (such as iterator expressions) can introduce variable declarations.

Chapter 13
Constraints

This chapter describes the definition of constraints on classes. A constraint is an invariant that must hold true for all instances (values) of a class. The properties of a constraint are described by an expression that is evaluated in the context of each instance.

13.1 POSITION IN ARCHITECTURE



13.1.1 Example

Figure 13-1 on page 158 shows an example of a simple constraint on a class A. It states that the attribute x in A must always be equal to 10 for all instances of A.

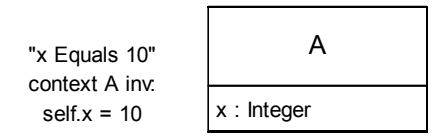


Figure 13-1 *An example of a constraint on a class*

13.2 ABSTRACT SYNTAX

13.2.1 Derivation

Figure 13-2 on page 158 describes how the constraints abstract syntax package is derived from the StructuralFeatureClassifier and ExpressionContext templates. A constraint is a structural feature. A constraint is associated with a static expression and has a type (which should be a boolean).

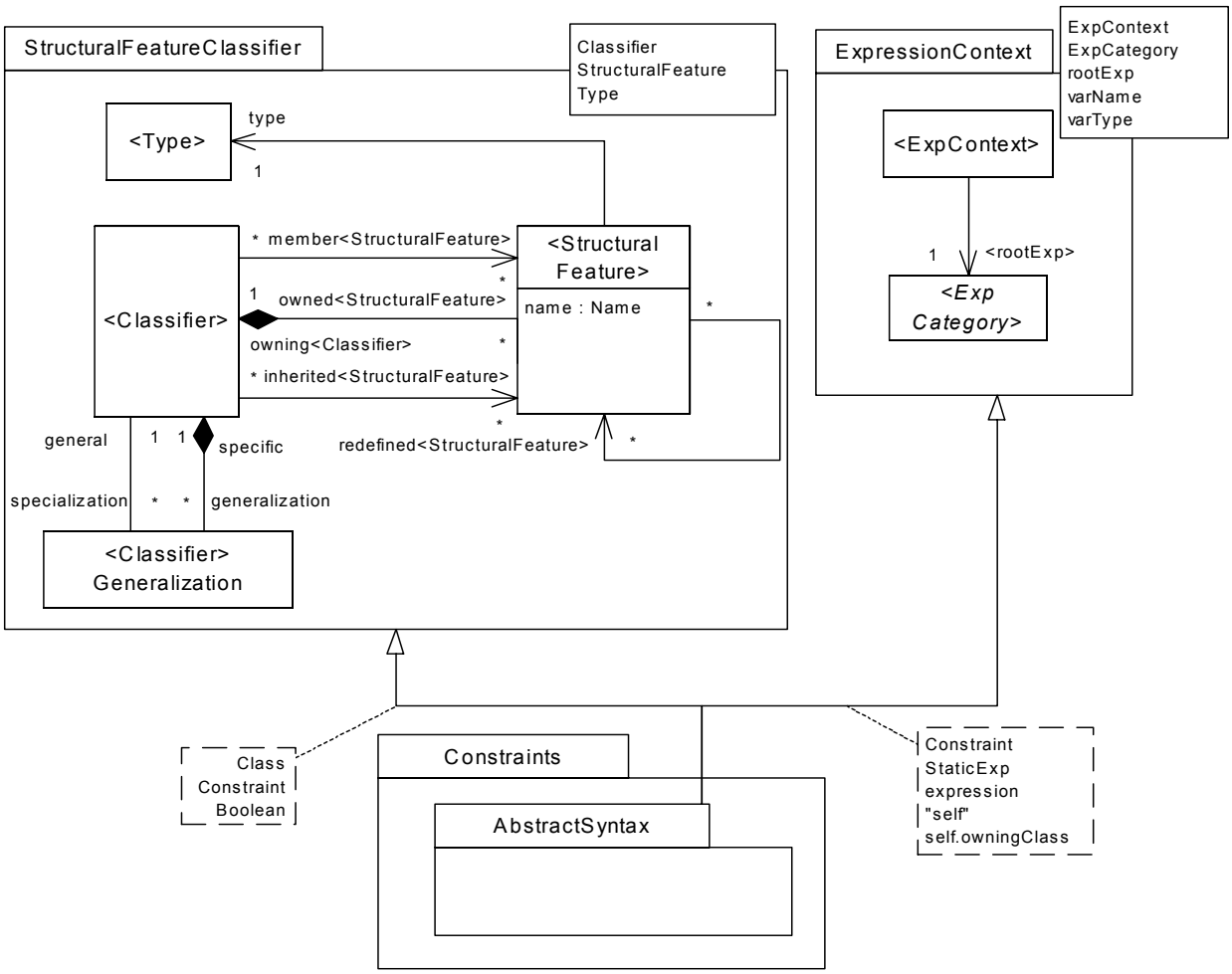


Figure 13-2 *Derivation of Constraints abstract syntax package.*

13.2.2 Model

Figure 13-3 on page 159 shows the abstract syntax for the constraints package. Classes are namespaces for constraints. Constraints have a name, an expression and a type. A generalisation relationship results in all constraints of the parent class being inherited by the child class (unless they are redefined).

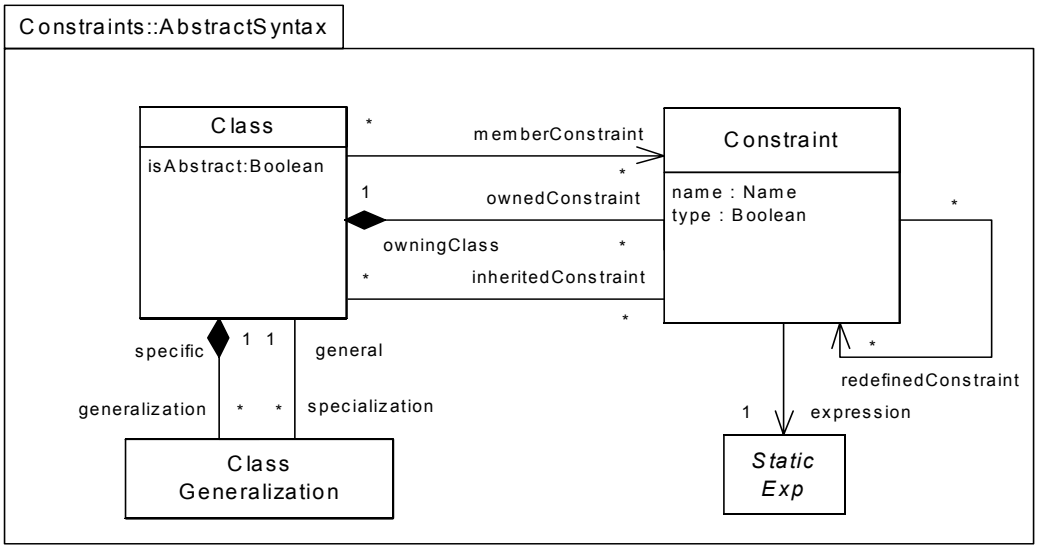


Figure 13-3 *Abstract syntax for the Constraints package.*

Class

A class is a namespace for its constraints.

Attributes

isAbstract Describes whether or not the class is abstract.

Associations

generalization The generalizations of the class.

inheritedConstraint The constraints inherited by the class.

memberConstraint The set of all constraints in the namespace of the class.

ownedConstraint The constraints owned by the class.

specializations The specialisations of the class.

Constraint

A constraint is an invariant property of a class that holds true for all values of the class. A constraint has an static expression that describes the properties of the constraint.

Attributes

name The name of the constraint.

Associations

expression The expression that describes the properties of the constraint.

owningClass The class that owns the constraint.

redefinedConstraint The constraints that have been redefined by the constraint.

type The type of the constraint.

ClassGeneralization

A generalization relationship between classes.

Associations

general The class that is the more general (parent) class in the relationship.

specific The class that is the more specific (child) class in the relationship.

StaticExpression

An abstract static expression. This class is specialised in Chapter 12 with concrete expressions.

13.2.3 Well-formedness Rules

Class

[1] The members of a class include its owned and inherited constraints.

```
context Class inv:
  self.memberConstraint->includesAll(self.ownedConstraint ->
    union(self.inheritedConstraint))
```

[2] Constraints cannot be owned and inherited.

```
context Class inv:
  self.ownedConstraint->intersection(self.inheritedConstraint) -> isEmpty
```

[3] A class cannot have two constraints with the same name.

```
context Class inv:
  self.memberConstraint->forAll(e1|
    self.memberConstraint->forAll(e2|
      e1 <> e2 implies e1.name <> e2.name))
```

[4] The inherited members of a class are the constraints of its parents classes that aren't redefined.

```
context Class inv:
  self.inheritedConstraint = self.generalElements()->iterate(p s = Set{} |
     s->union(p.memberConstraint->reject(c |
       self.memberConstraint -> exists(c' |
         c'.redefinedConstraint->includes(c)))))
```

[5] A class's constraints may only redefine its parent classes constraints.

```
context Class inv:
  self.memberConstraint -> forAll(a |
    self.generalElements()-> collect(g | g.memberConstraint) ->
      includesAll(a.redefinedConstraint))
```

Constraint

[1] A constraint introduces the variable declaration "self" into its scope.

```
context Constraint inv:
  self.expression.scope -> exists(v | v.varName="self" and
    v.type=self.owningClass)
```

13.2.4 Operations

Class

[1] Looks up a constraint in a class when given a name.

```
context Class::lookupConstraintforName(x : Name):Constraint
  self.memberConstraint->select(e| e.name = x ).selectElement()
```

[2] Looks up a constraint's name when given the constraint.

```
context Class::lookupNameForConstraint(x : Constraint):Name
  self.memberConstraint->select(e|e = x ).selectElement().name
```

13.3 SEMANTIC DOMAIN

13.3.1 Derivation

Figure 13-4 on page 161 shows how the Constraints semantic domain package is derived from the StructuralFeatureClasifierValue and ExpressionContextValue templates. A constraint evaluation is structural feature value and has an expression evaluation that is evaluated in the context of its owning object.

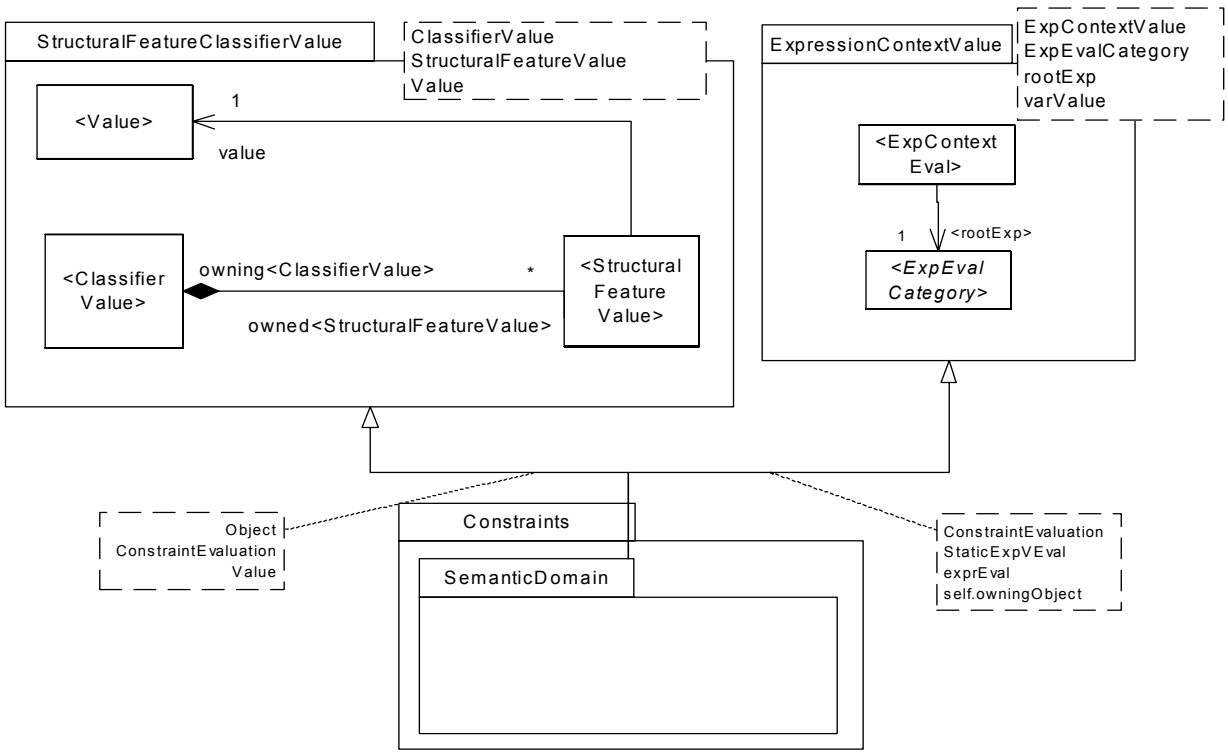


Figure 13-4 *Derivation of Constraints semantic domain package*

13.3.2 Model

Figure 13-5 on page 162 shows the semantic domain of the constraints package. A constraint evaluation describes the result of evaluating a static expression. The result must be true in the context of the constraint evaluation's owning object (the object that is bound to the variable "self").

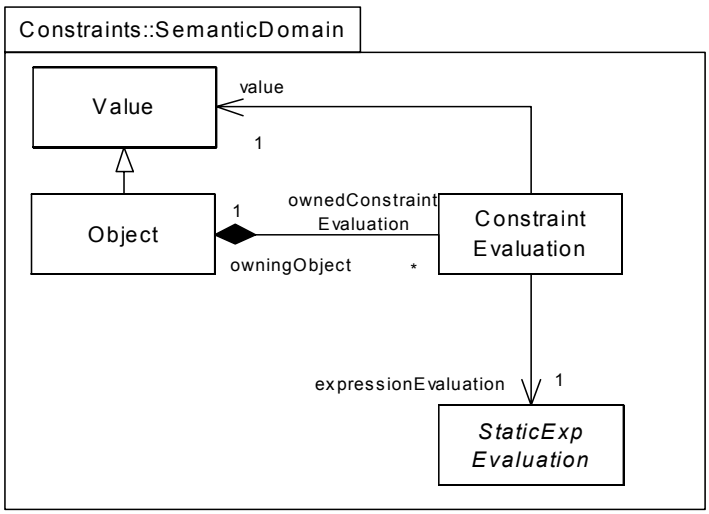


Figure 13-5 *Semantic domain for the Constraints package*

ConstraintEvaluation

Constraint evaluations describe the result of evaluating an expression belonging to a constraint.

Associations

expressionEvaluation A constraint's expression evaluation.

owningObject The object that is the context of the constraint evaluation.

value The result of the constraint evaluation.

13.3.3 Well-formedness Rules

ConstraintEvaluation

[1] A constraint evaluation introduces the value of its context into the environment of its expression evaluation.

```
context ConstraintEvaluation inv:
  self.expressionEvaluation.env -> exists(v |
    v.value=self.owningObject)
```

[2] A constraint evaluation's value should be the same as its expression evaluation's value.

```
context ConstraintEvaluation inv:
  self.value = self.expressionEvaluation.value
```

[3] A constraint evaluation's value should evaluate to true.

```
context ConstraintEvaluation inv:
  self.value = true
```

13.4 SEMANTIC MAPPING

13.4.1 Derivation

Figure 13-6 on page 163 illustrates the derivation of the Constraints semantic mapping package using the StructuralFeatureClassifierSemantics template.

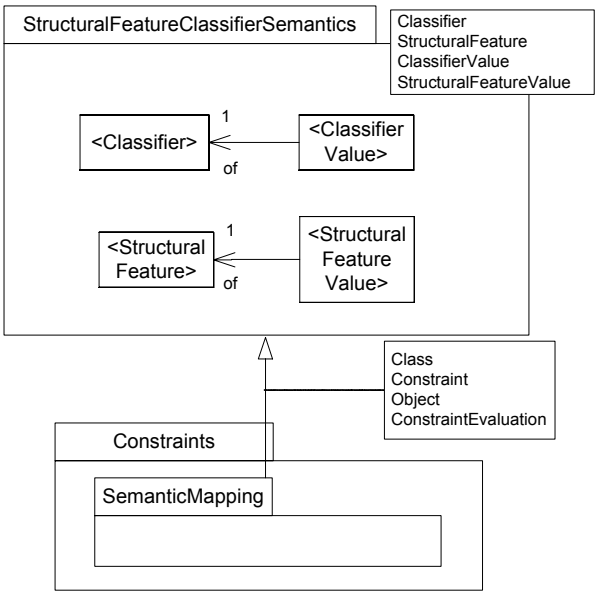


Figure 13-6 *Derivation of Constraints semantic mapping package*

13.4.2 Model

The semantic mapping for the Constraints package is shown in figure 13-7 on page 163. An expression evaluation is a value of an expression and must contain a variable value that binds the variable "self". An object must contain a constraint evaluation for each of its class's constraints and vice versa.

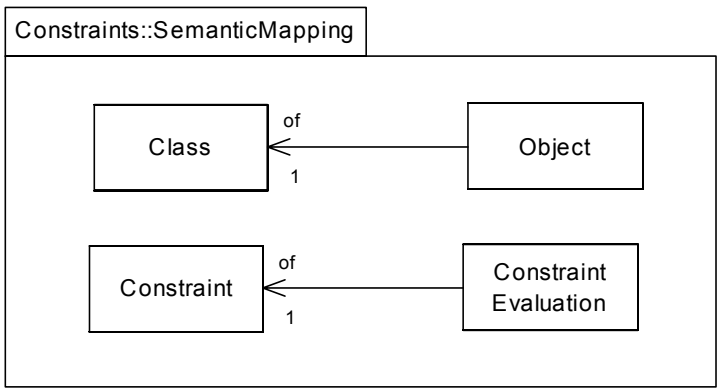


Figure 13-7 *Semantic mapping for Constraints package*

ConstraintEvaluation

Associations

of The constraint of which the constraint evaluation is a value.

13.4.3 Well-formedness rules

ConstraintEvaluation

[1] A constraint evaluation will bind the variable value "self" to its owning object.

```
context ConstraintEvaluation inv:
  self.expressionEvaluation.env -> forAll(v |
    v.of.varName="self" implies v.value=self.owningObject)
```

[2] An expression evaluation's expression commutes with its constraint's expression.

```
context ConstraintEvaluation inv:
  self.expressionEvaluation.of = self.of.expression
```

[3] The value of a constraint evaluation should be a value of the type that conforms to its constraint's type.

```
context ConstraintEvaluation inv:
  self.value.of.conformsTo(self.of.type)
```

Object

[1] An object should contain a constraint evaluation for all constraints in the object's class's namespace.

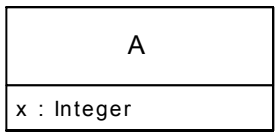
```
context Object inv:
  self.of.memberConstraint->forAll(c |
    self.ownedConstraintEvaluation->exists(d | d.of = c))
```

[2] For each constraint evaluation owned by an object there should be an constraint of the object's class's namespace that the constraint evaluation is a value of.

```
context Object inv:
  self.ownedConstraintEvaluation->forAll(c |
    self.of.memberConstraint->exists(d | c.of = d))
```

13.5 EXAMPLE SNAPSHOTS

Figure 13-9 on page 165 shows a partial snapshot of the evaluation of the constraint shown in figure 13-8 on page 164. The complete evaluation of the expression is omitted for brevity. A constraint is satisfied if it evaluates to true in the context of an instance of its class. Note how the scope of the equals evaluation expression binds the constrained object to the variable "self".



"x Equals 10"
context A inv:
self.x = 10

Figure 13-8 *Example class and constraint*

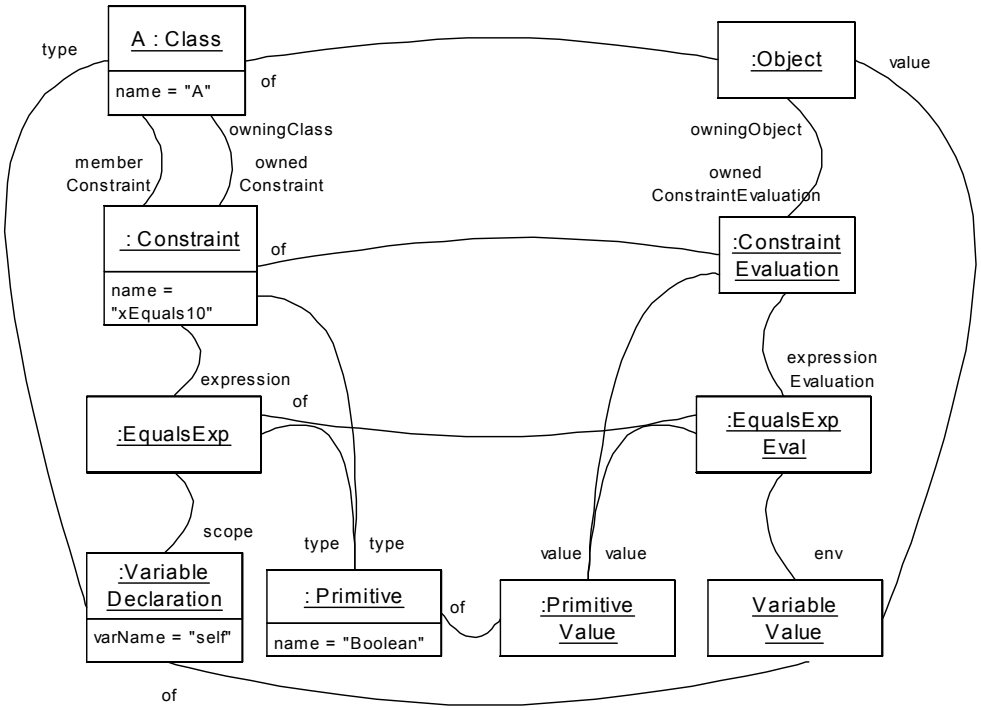


Figure 13-9 *Snapshot of Constraints semantic mapping package*

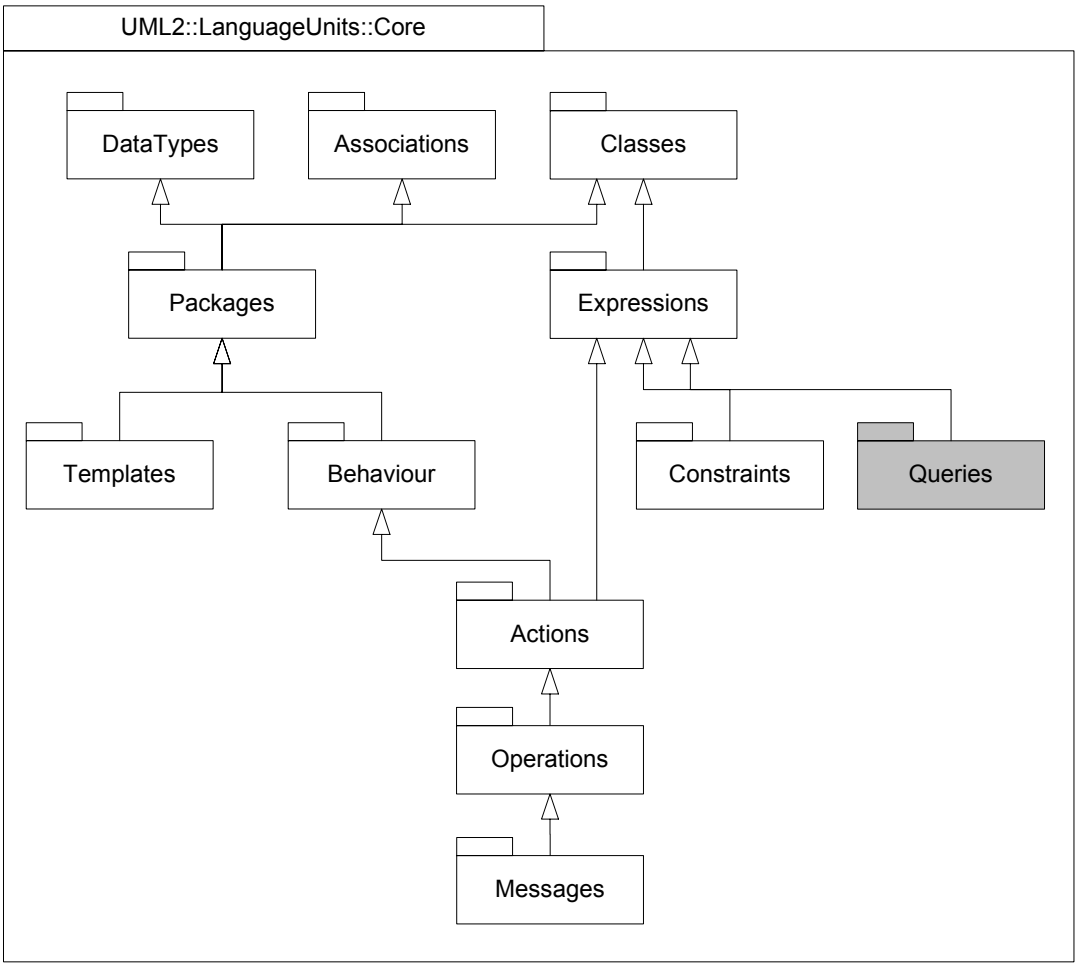
13.6 CHANGES TO UML 1.4

In UML 1.4, constraint is a concrete class that can be applied to any model element. The machinery involved in evaluating a constraint for any model element is unacceptably vague in UML 1.4 given the importance of constraints in the definition of UML itself. In this submission, templates for defining and evaluating expressions can be used to generate context specific constraints on any type of element. However, class constraints are considered sufficient for the infrastructure submission due to the fact that they are the most widely used constraint in UML.

Chapter 14
Queries

This chapter describes the definition of queries. A query is a static operation that returns a result in the context of an instance of a class. The properties of a query are described by an expression.

14.1 POSITION IN ARCHITECTURE



14.1.1 Example

Figure 14-1 on page 167 shows an example of a simple query on a class A. It returns the value of the attribute x plus the value of the passed parameter variable y.

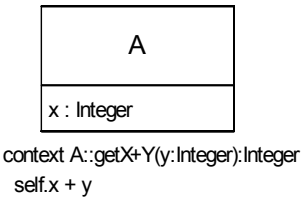


Figure 14-1 *An example of a query on a class*

14.2 ABSTRACT SYNTAX

Figure 14-2 on page 168 describes how the queries abstract syntax package is derived from the StructuralFeature-Classifier, ExpressionContext and Parameterized templates. A query is a structural feature. A query is also a parameterized element and is associated with a static expression.

14.2.1 Derivation

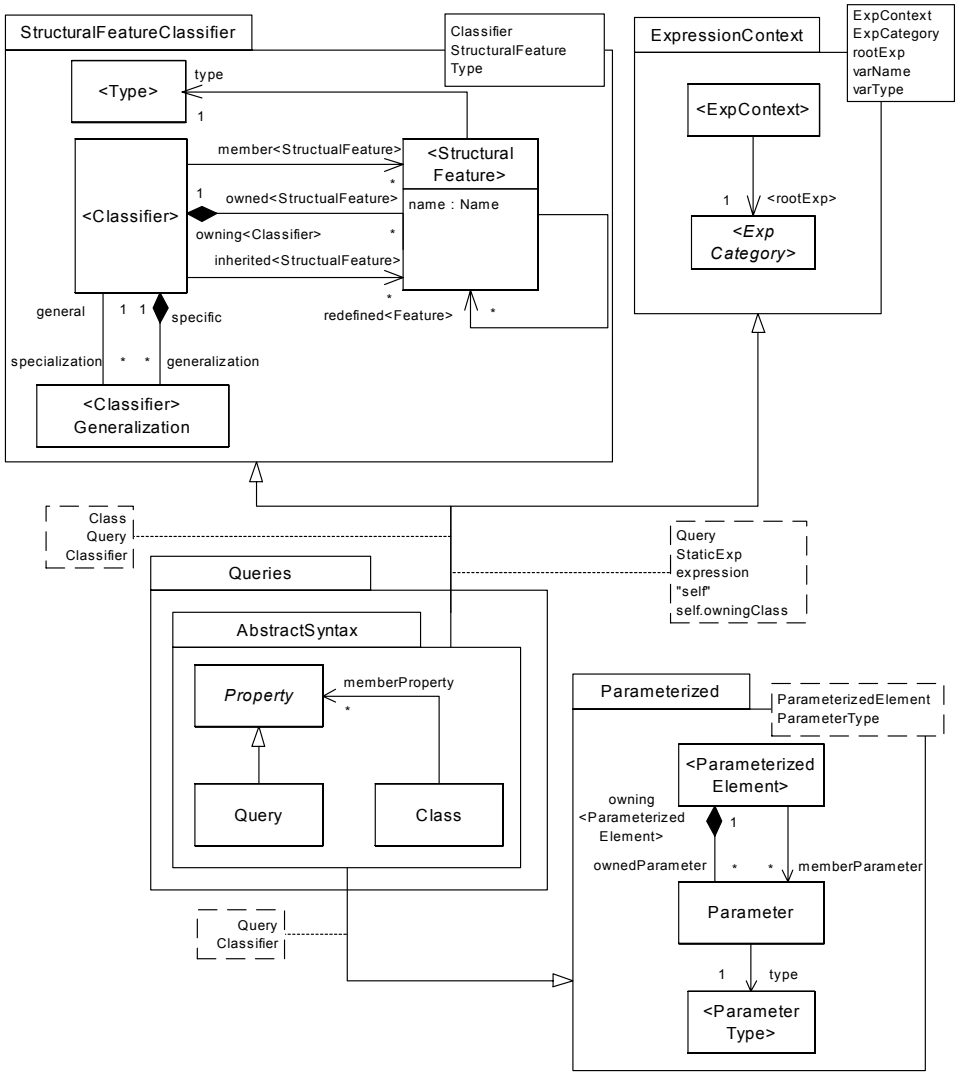


Figure 14-2 *Derivation of Queries abstract syntax package.*

14.2.2 Model

Figure 14-3 on page 169 shows the abstract syntax for the queries package. Classes are namespaces for queries. Queries have a name, a type, an expression and a set of parameters. A generalisation relationship results in all queries of the parent class being inherited by the child class (unless they are redefined).

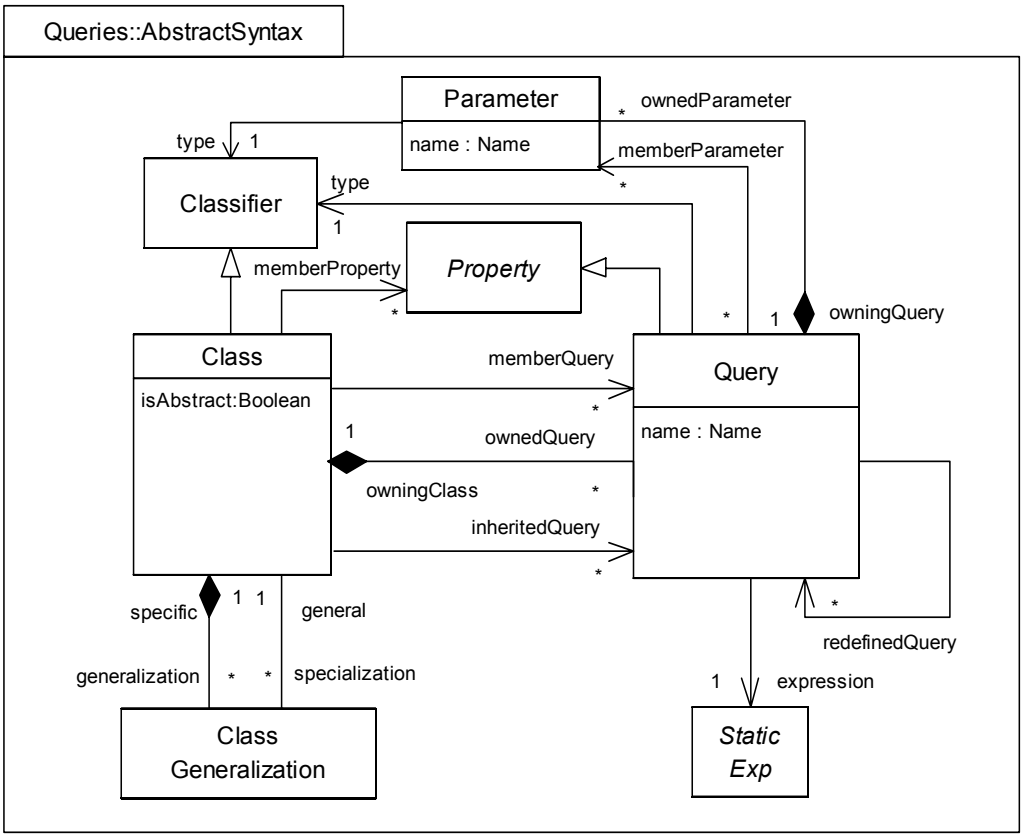


Figure 14-3 *Abstract syntax for the Queries package.*

Class

A class is a namespace for its queries.

Attributes

isAbstract Describes whether or not the class is abstract.

Associations

generalization The generalizations of the class.

inheritedQuery The queries inherited by the class.

memberQuery The set of all queries in the namespace of the class.

ownedQuery The queries owned by the class.

specialization The specializations of the class.

memberProperty The properties that are a member of the class.

Query

A query is a static operation that returns a value in the context of an instance of a class. A query has a static expression that describes the result of the query. A query is a property, and can therefore be accessed through a property call expression (see Chapter ***).

Attributes

name The name of the query.

Associations

expression The expression that describes the result of the query.

memberParameter The parameters in the namespace of the query.

ownedParameter The parameters owned by the query.

owningClass The class that owns the query.

redefinedQuery The queries that have been redefined by the query.

type The return type of the query.

ClassGeneralization

A generalization relationship between classes.

Associations

general The class that is the more general (parent) class in the relationship.

specific The class that is the more specific (child) class in the relationship.

StaticExpression

An abstract static expression. This class is specialised in Chapter 11 with concrete expressions.

14.2.3 Well-formedness Rules

Class

[1] The member queries of a class include its owned and inherited queries.

```
context Class inv:
  self.memberQuery->includesAll(self.ownedQuery ->
    union(self.inheritedQuery))
```

[2] Queries cannot be owned and inherited.

```
context Class inv:
  self.ownedQuery->intersection(self.inheritedQuery) -> isEmpty
```

[3] A class cannot have two queries with the same name.

```
context Class inv:
  self.memberQuery->forAll(e1|
    self.memberQuery->forAll(e2|
      e1 <> e2 implies e1.name <> e2.name))
```

[4] The inherited members of a class are the queries of its parents classes that are not redefined.

```
context Class inv:
  self.inheritedQuery = self.generalElements()->iterate(p s = Set{} |
     s->union(p.memberQuery->reject(c |
       self.memberQuery -> exists(c' |
         c'.redefinedQuery->includes(c)))))
```

[5] A class's queries may only redefine its parent classes queries.

```
context Class inv:
  self.memberQuery -> forAll(a |
    self.generalElements()-> collect(g | g.memberQuery) ->
      includesAll(a.redefinedQuery))
```

[6] A class's member properties include its member queries.

```
context Class inv:
  self.memberProperty -> includesAll(memberQuery)
```

Query

[1] A query's type must conform to the type of its redefined queries.

```
context Query inv:
  self.redefinedQuery->forAll(f |
    self.type.conformsTo(f.type))
```

[2] The members of a query include its owned parameters

```
context Query inv:
  self.memberParameter->includesAll(self.ownedParameter)
```

[3] A query cannot have two parameters with the same name.

```
context Query inv:
  self.memberParameter->forAll(e1|
    self.memberParameter->forAll(e2|
      e1 <> e2 implies e1.name <> e2.name))
```

[4] A query introduces the variable declaration "self" into its scope.

```
context Query inv:
  self.expression.scope -> exists(v | v.varName="self" and
    v.type=self.owningClass)
```

[5] A query introduces variable declarations for each of its parameters into its scope.

```
context Query inv:
  self.parameter -> forAll(p |
    self.expression.scope -> exists(v | v.varName=p.name and
      v.type=p.type))
```

14.2.4 Operations

Class

[1] Looks up a query in a class when given a name.

```
context Class::lookupQueryforName(x : Name):Query
  self.memberQuery->select(e| e.name = x ).selectElement()
```

[2] Looks up a query's name when given the query.

```
context Class::lookupNameForQuery(x : Query):Name
  self.memberQuery->select(e|e = x ).selectElement().name
```

Query

[1] Looks up a parameter in a query when given a name.

```
context Query::lookupParameterforName(x : Name):Parameter
  self.memberParameter->select(e| e.name = x ).selectElement()
```

[2] Looks up a parameter's name when given the parameter.

```
context Query::lookupNameForParameter(x : Parameter):Name
  self.memberParameter->select(e|e = x ).selectElement().name
```

14.3 SEMANTIC DOMAIN

14.3.1 Derivation

Figure 14-4 on page 172 describes how the Queries semantic domain package is derived from the Expression-ContextValue template. A query evaluation is an expression evaluation that is evaluated in the context of an object.

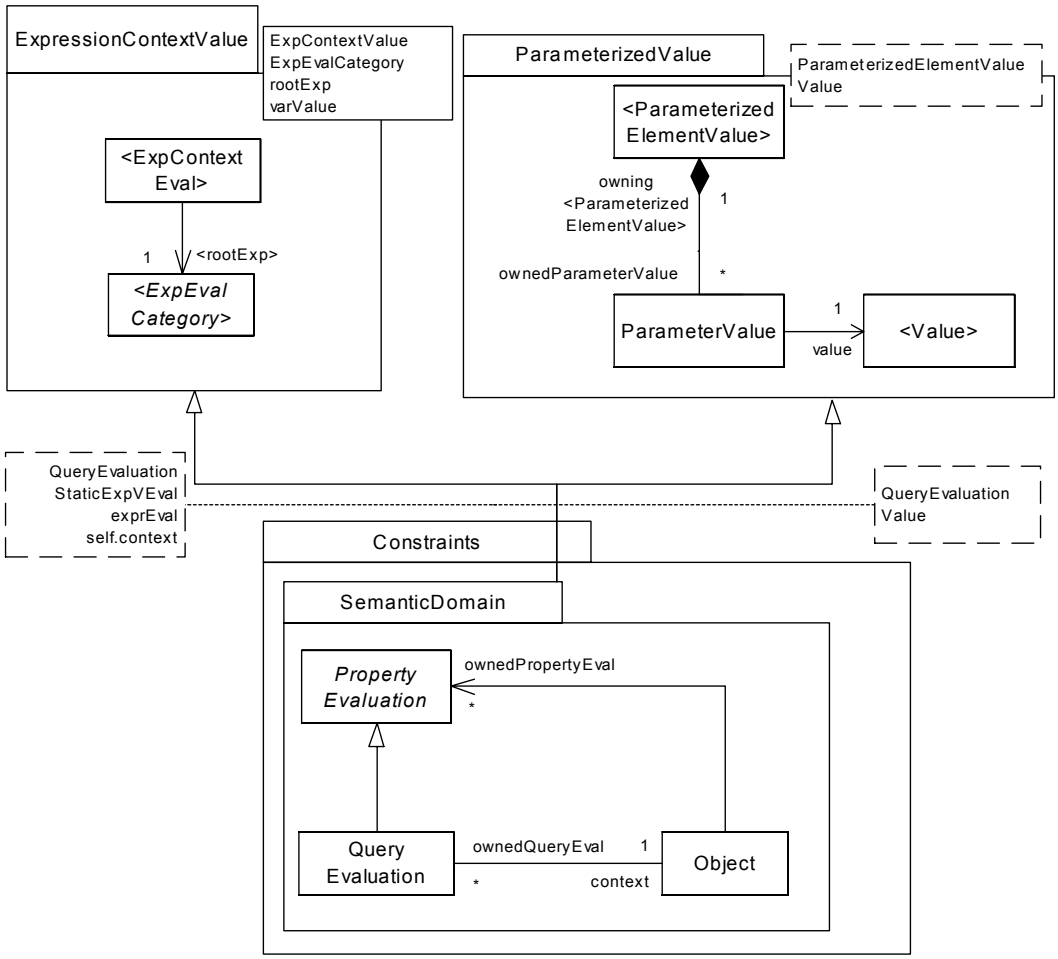


Figure 14-4 *Derivation of Queries semantic domain package*

14.3.2 Model

Figure 14-5 on page 173 shows the semantic domain of the queries package. A query evaluation describes the result of evaluating a static expression. The result is calculated in the context of the query evaluation's context (the object that is bound to the variable "self") and its bound parameter values.

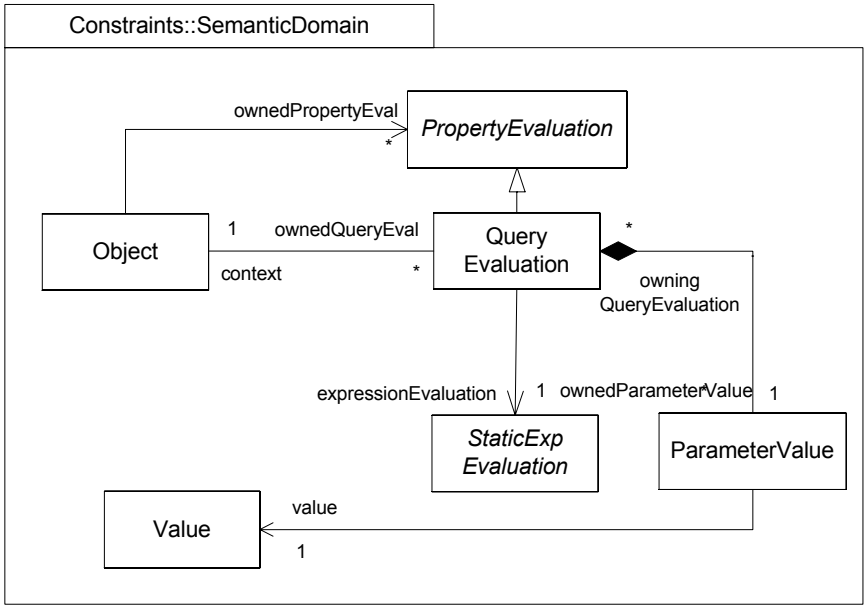


Figure 14-5 *Semantic domain for the Queries package*

QueryEvaluation

Query evaluations describe the result of evaluating an expression belonging to a query. A query evaluation is a property evaluation, which means it can be referenced through a property call evaluation (see Expressions chapter).

Associations

context The object that is the context of the query evaluation.

expressionEvaluation A query's expression evaluation.

ownedParameterValue The parameter values owned by the query evaluation.

Object

Associations

ownedPropertyEval The property evaluations owned by the object.

ownedQueryEval The query evaluations owned by the object.

ParameterValue

Associations

owningQueryEvaluation The query evaluation owning the parameter.

14.3.3 Well-formedness Rules

QueryEvaluation

[1] A query evaluation introduces the value of its context into the environment of its expression evaluation.

```
context QueryEvaluation inv:
  self.expressionEvaluation.env -> exists(v |
    v.value=self.context)
```

[2] A query evaluation introduces the value of its parameters into the environment of its expression evaluation.

```
context QueryEvaluation inv:
  self.ownedParameterValue -> forAll(p |
    self.expressionEvaluation.scope -> exists(v |
      v.value=p.value))
```

14.4 SEMANTIC MAPPING

Figure 14-7 on page 175 defines the semantic mapping of the Queries package. The satisfaction of a query evaluation's is determined by its expression evaluation.

14.4.1 Derivation

Figure 14-6 on page 174 illustrates the derivation of the Queries semantic mapping package using the Semantics template. An expression evaluation is an instance of an expression and must contain a variable value that binds the variable "self".

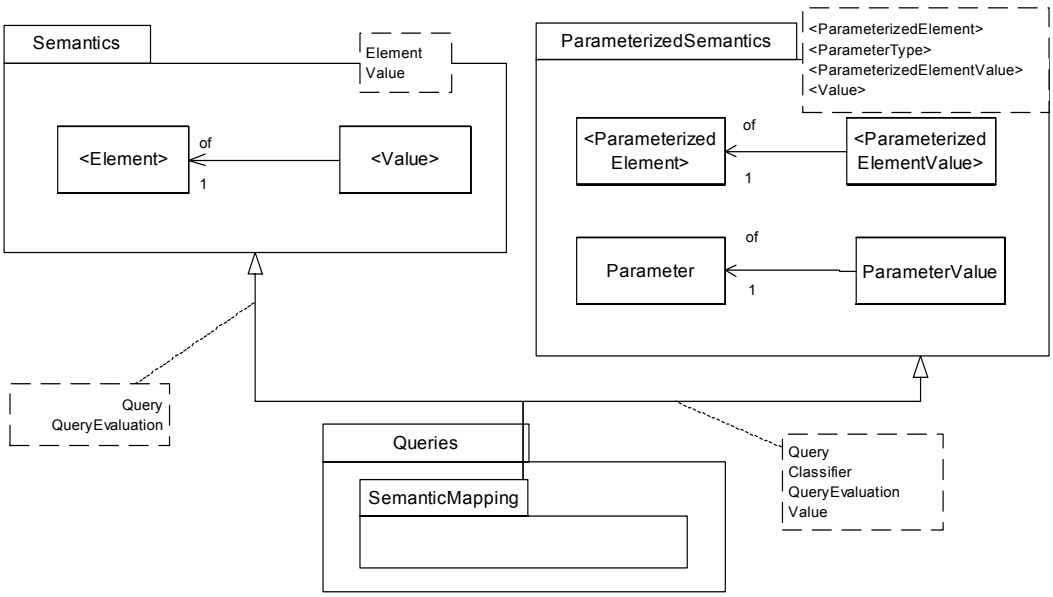


Figure 14-6 *Derivation of Queries semantic mapping package*

14.4.2 Model

The semantic mapping for the Queries package is shown in figure 14-7 on page 175. A query evaluation is a value of a query. A parameter value is a value of a parameter.

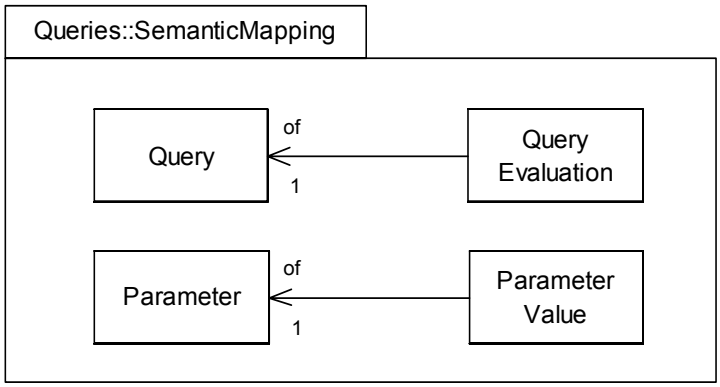


Figure 14-7 *Semantic mapping for Queries package*

QueryEvaluation

Associations

of The query of which the query evaluation is a value.

ParameterValue

Associations

of The parameter of which the parameter value is a value.

14.4.3 Well-formedness rules

Object

[1] For each property evaluation owned by an object there should be a property of the object's class's namespace that the property is a value of.

```
context Object inv:
  self.ownedPropertyEvaluation->forAll(pv |
    self.of.memberProperty->exists(p | pv.of = p))
```

QueryEvaluation

[1] Ensures that the variable value bound to self is the context of the query evaluation.

```
context QueryEvaluation inv:
  self.expressionEvaluation.env -> forAll(v |
    v.of.varName="self" implies v.value=self.context)
```

[2] Ensures that the variables values bound to parameter names are the parameter values of the query evaluation.

```
context QueryEvaluation inv:
  self.parameterValue -> forAll(p |
    self.expressionEvaluation.env -> forAll(v |
      v.of.varName=p.of.name implies v.value=p.value)
```

[3] The query evaluation's expression evaluation commutes with its query's expression.

```
context QueryEvaluation inv:
  self.expressionEvaluation.of = self.of.expression
```

[4] A query evaluation should contain a parameter value for all parameter's in the query evaluation's query's namespace.

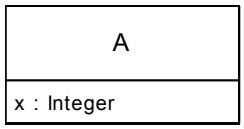
```
context QueryEvaluation inv:
  self.of.memberParameter->forAll(c |
    self.ownedParameterValue->exists(d | d.of = c))
```

[5] For each parameter value owned by a query evaluation there should be a parameter of the query evaluation's query's namespace that the parameterized element value is a value of.

```
context QueryEvaluation inv:
  self.ownedParameterValue->forAll(c |
    self.of.memberParameter->exists(d | c.of = d))
```

14.5 SNAPSHOT

Figure 14-9 on page 177 shows a partial snapshot of the evaluation of the query shown in figure 14-8 on page 176. The complete evaluation of the expression is omitted for brevity. A query evaluation's evaluation expression returns a value in the context of an instance of its class and a collection of bound parameter variables.




```
context A::getX+Y(y:Integer):Integer
  self.x + y
```

Figure 14-8 *Example class and query*

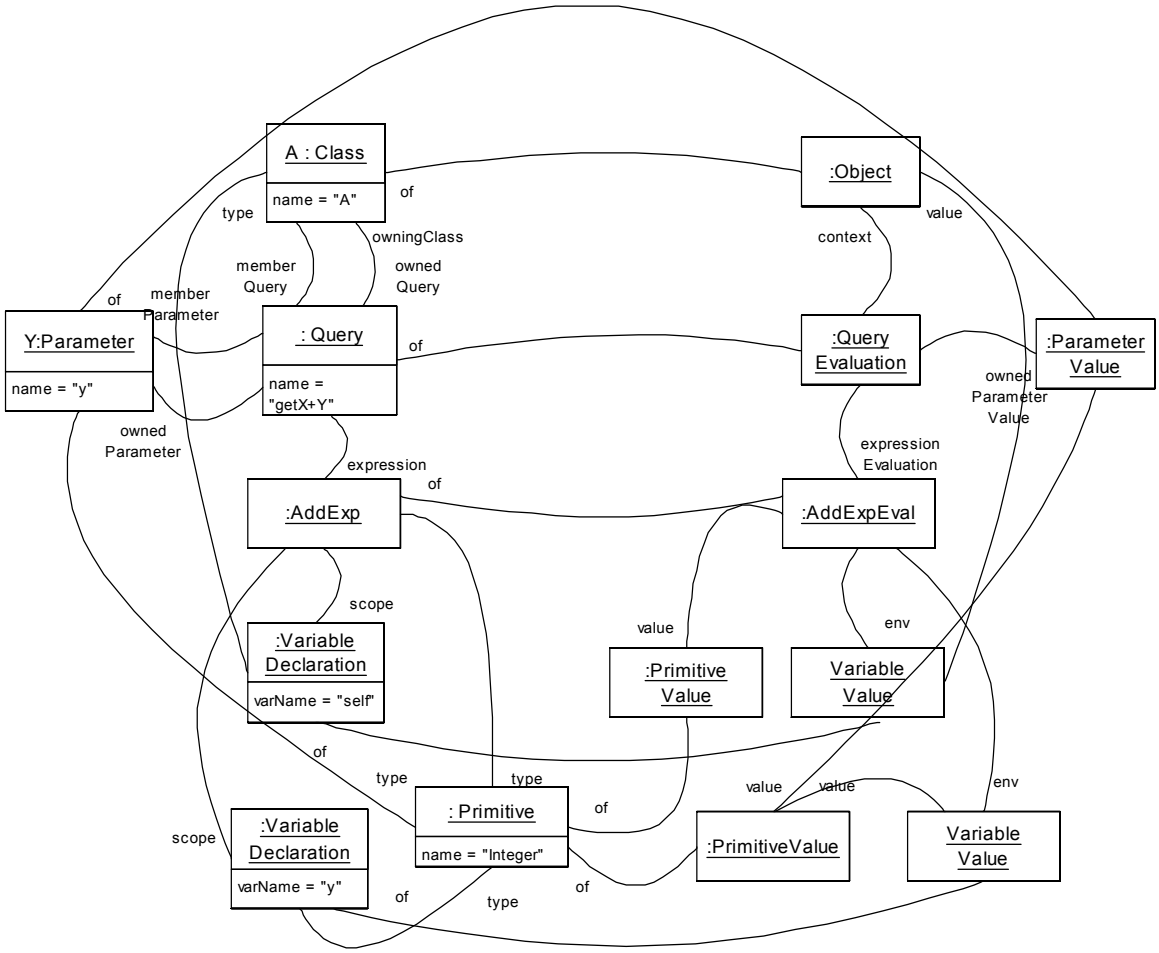


Figure 14-9 *Snapshot of Queries semantic mapping package*

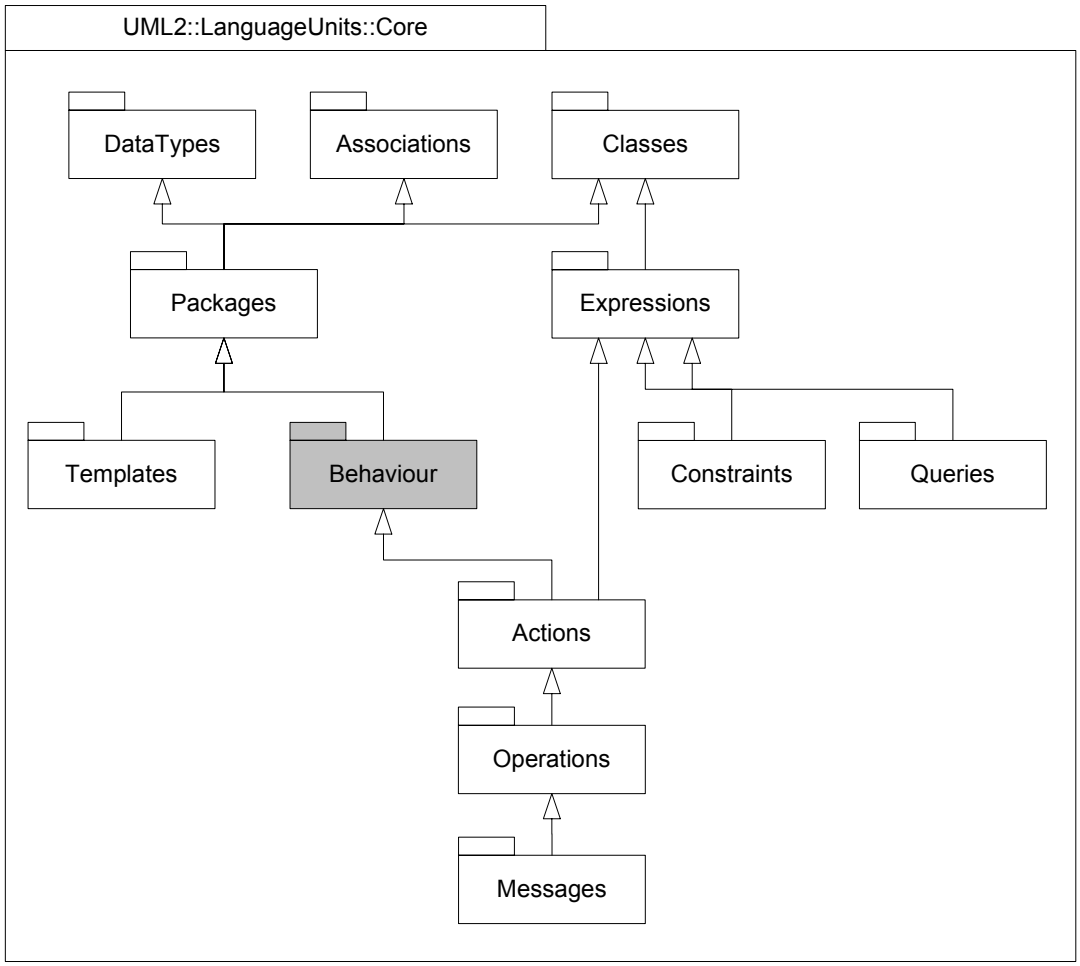
14.6 CHANGES TO UML 1.4

UML 1.4 defines a query as an operation with isQuery set to true. However, the semantics of queries are static, and not operational, and therefore it makes sense to define them as a stand-alone static concept.

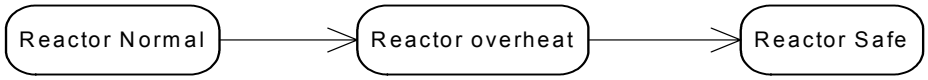
Chapter 15
Behaviour

The definition described so far has been concerned with characterising the static components of UML. In this chapter we describe the behaviour package which deals with supporting the modelling of systems which evolve over time. This is achieved by enabling the instances of model elements to have multiple states at different points in time. These states are related by the ordering in which they occurred and a mechanism that manages this ordering. The definition presented here lays a foundation for the definition of actions (Chapter 16) and operations (Chapter 17).

15.1 POSITION IN ARCHITECTURE



15.1.1 Example



15.2 ABSTRACT SYNTAX

Figure 15-1 on page 179 shows the abstract syntax for the Behaviour package. A package has member packages and member classes, and classes have member attributes.

15.2.1 Model

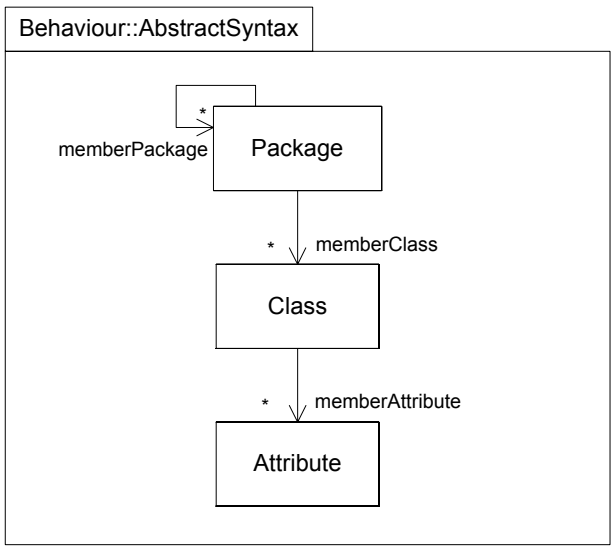


Figure 15-1 *Abstract Syntax for Behaviour package*

Package

Associations

memberPackage The member packages.

memberClass The member classes.

Class

Associations

memberAttributes The member attributes.

15.2.2 Well-formedness Rules

There are no well-formedness rules.

15.2.3 Operations

There are no operations.

15.3 SEMANTIC DOMAIN

15.3.1 Derivation

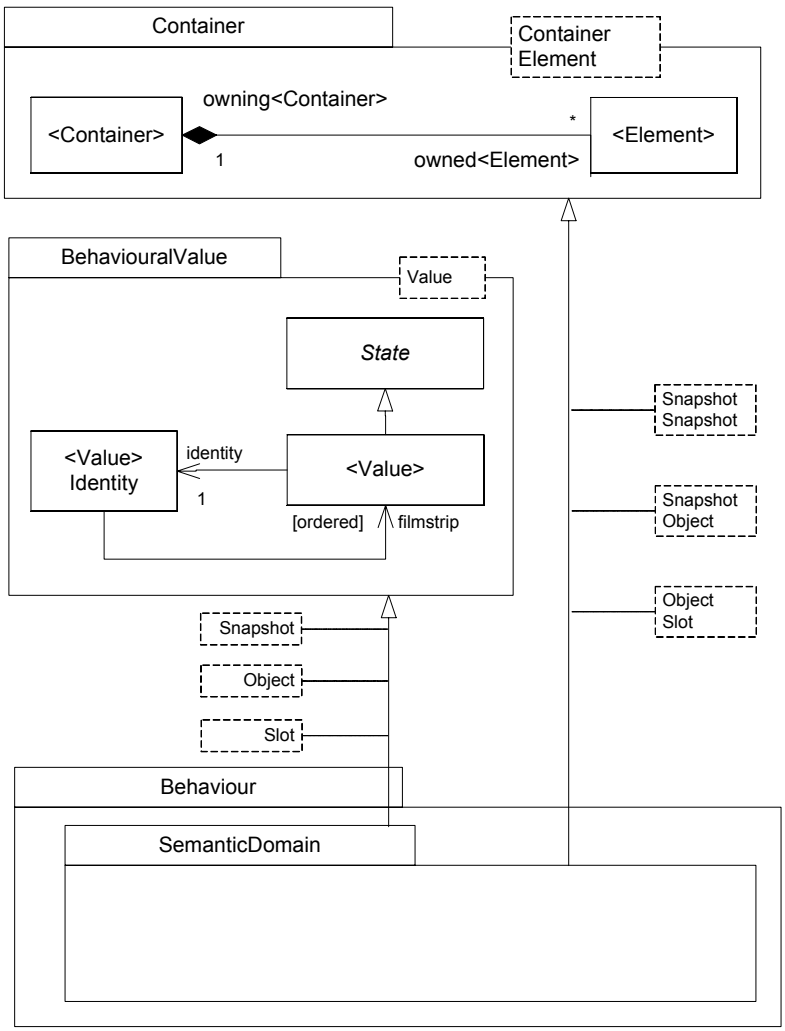


Figure 15-2 *Derivation of Behaviour Semantic Domain package*

15.3.2 Model

Figure 15-3 on page 181 shows the semantic domain for the Behaviour packages derived as illustrated in figure 15-2 on page 180. A snapshot has an identity, and the identity has a set of snapshots ordered in a filmstrip. An object has an identity, and the identity has a set of objects ordered in a filmstrip. Similarly, a slot has an identity, and the identity has a set of slots ordered in a filmstrip. An identity can be considered as persisting through time, whereas the elements ordered by the identity's filmstrip (i.e. snapshots,objects and slots) are the same element at different periods of time.

A snapshot contains snapshots and objects, and an object contain slots. A snapshot, object and slot are generalised from State. A State can be considered as the state of an element at a particular time frame.

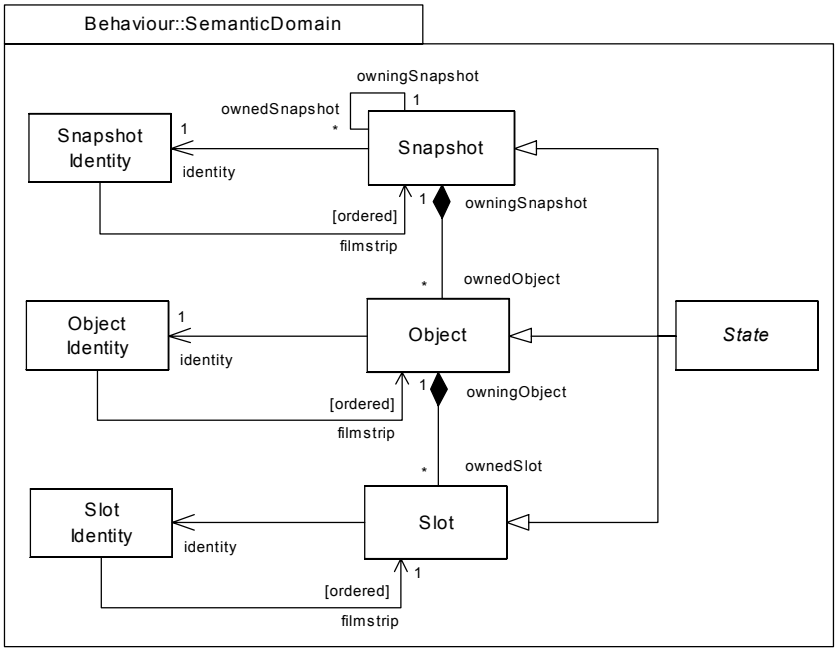


Figure 15-3 *Semantic Domain for Behaviour package*

Snapshot

Associations

identity The identity of the snapshot.

ownedSnapshot The snapshots owned by the snapshot.

ownedObject The objects owned by the snapshot.

owningSnapshot The snapshot owning the snapshot.

SnapshotIdentity

Associations

filmstrip An ordered set of snapshots.

Object

Associations

identity The identity of the object.

ownedSlot The slots owned by the object.

owningSnapshot The snapshot owning the object.

ObjectIdentity

Associations

filmstrip An ordered set of objects.

Slot

Associations

identity The identity of the slot.

owningObject The object owning the slot.

SlotIdentity

Associations

filmstrip An ordered set of slots.

15.3.3 Well-formedness Rules

SnapshotIdentity

[1] The identity of the snapshot commutes with its filmstrips.

```
context SnapshotIdentity inv:
  self.filmstrip->forAll(v | v.identity = self)
```

[2] Each snapshot in the filmstrip must be unique.

```
context SnapshotIdentity inv:
   self.flimstrip->forAll(e1 | self.filmstrip->forAll(e2 | e1 <> e2))
```

ObjectIdentity

[1] The identity of the object commutes with its filmstrips.

```
context ObjectIdentity inv:
  self.filmstrip -> forAll(v | v.identity = self)
```

[2] Each object in the filmstrip must be unique.

```
context ObjectIdentity inv:
  self.flimstrip->forAll(e1 | self.filmstrip->forAll(e2 | e1 <> e2))
```

SlotIdentity

[1] The identity of the slot commutes with its filmstrips.

```
context SlotIdentity inv:
    self.filmstrip -> forAll(v | v.identity = self)
```

[2] Each slot in the filmstrip must be unique.

```
context SlotIdentity inv:
  self.flimstrip->forAll(e1 | self.filmstrip->forAll(e2 | e1 <> e2))
```

15.3.4 Operations

Absolute ordering of states is maintained by the filmstrip of the root snapshot identity. This contains a number of operations which enable the comparison of the temporal occurrence of two states (snapshots, objects or slots). Each state has an operation, such as *isLater*, which given a state checks to see where that state occurs. This is achieved by navigating to the root snapshot's identity and calling the namesake operation, such as *isLater*, with the state and self. Each state also has an operation (*isState*) which checks to see if two states are in fact the same state. This is used by the root snapshot identity in determining where states occur within its filmstrip.

SnapshotIdentity

[1] Given two states determines whether the first state occurs before the second.

```
context SnapshotIdentity::isEarlier(s1:State,s2:State):Boolean
  state1:State
  state2:State
  filmstrip->forAll(s | if(s.isState(s1)) state1 = s
                        if(s.isState(s2)) state2 = s)
  filmstrip.getIndex(state1) < filmstrip.getIndex(state2)
```

[2] Given two states determines whether the first state occurs after the second.

```
context SnapshotIdentity::isLater(s1:State,s2:State):Boolean
  state1:State
  state2:State
  filmstrip->forAll(s | if(s.isState(s1)) state1 = s
                        if(s.isState(s2)) state2 = s)
  filmstrip.getIndex(state1) > filmstrip.getIndex(state2)
```

[3] Given two state determines whether the first state occurs at the same time as the second.

```
context SnapshotIdentity::isSameTime(s1:State,s2:State):Boolean
  state1:State
  state2:State
  filmstrip->forAll(s | if(s.isState(s1)) state1 = s
                        if(s.isState(s2)) state2 = s)
  filmstrip.getIndex(state1) = filmstrip.getIndex(state2)
```

Snapshot

[1] Checks to see if the snapshot, or any of its owned objects or snapshots, are the same as a given state.

```
context Snapshot::isState(s: State):Boolean
  flag: Boolean
  flag := false
  if(self = s)
    true
  else
    self.ownedSnapshot->forAll(i | if(i.isState(s)) flag := true)
    self.ownedObject->forAll(i | if(i.isState(s)) flag := true)
    flag
  end
```

[2] Checks to see if a state occurs before this snapshot.

```
context Snapshot::isEarlier(s:State):Boolean
  if(owningSnapshot<>self)
    owningSnapshot.isLater(s)
  else
    owningSnapshotIdentity(s,self)
  end
```

[3] Checks to see if a state occurs after this snapshot.

```
context Snapshot::isLater(s:State):Boolean
  if(owningSnapshot<>self)
    owningSnapshot.isLater(s)
  else
    owningSnapshotIdentity(s,self)
  end
```

[4] Checks to see if a state occurs at the same time as this snapshot.

```
context Snapshot::isSameTime(s:State):Boolean
  if(owningSnapshot<>self)
    owningSnapshot.isLater(s)
  else
    owningSnapshotIdentity(s,self)
  end
```

Object

[1] Checks to see if the object, or its slots, are the same as a given state.

```
context Object::isState(s:State):Boolean
  flag: Boolean
  flag := false
  if(self = s)
    true
  else
    self.ownedSlot->forAll(i | if(i.isState(s)) flag := true)
    flag
  end
```

[2] Checks to see if a state occurs before this object.

```
context Object::isEarlier(s:State):Boolean
  owningSnapshot.isEarlier(s)
```

[3] Checks to see if a state occurs after this snapshot.

```
context Object::isLater(s:State):Boolean
  owningSnapshot.isLater(s)
```

[4] Checks to see if a state occurs at the same time as this snapshot.

```
context Object::isSameTime(s:State):Boolean
  owningSnapshot.isSameTime(s)
```

Slot

[1] Checks to see if the slot is the same as a given state.

```
context Object::isState(s:State):Boolean
  self = s
```

[2] Checks to see if a state occurs before this object.

```
context Slot::isEarlier(s:State):Boolean
  owningSnapshot.isEarlier(s)
```

[3] Checks to see if a state occurs after this snapshot.

```
context Slot::isLater(s:State):Boolean
  owningSnapshot.isLater(s)
```

[4] Checks to see if a state occurs at the same time as this snapshot.

```
context Slot::isSameTime(s:State):Boolean
  owningSnapshot.isSameTime(s)
```

15.4 SEMANTIC MAPPING

15.4.1 Derivation

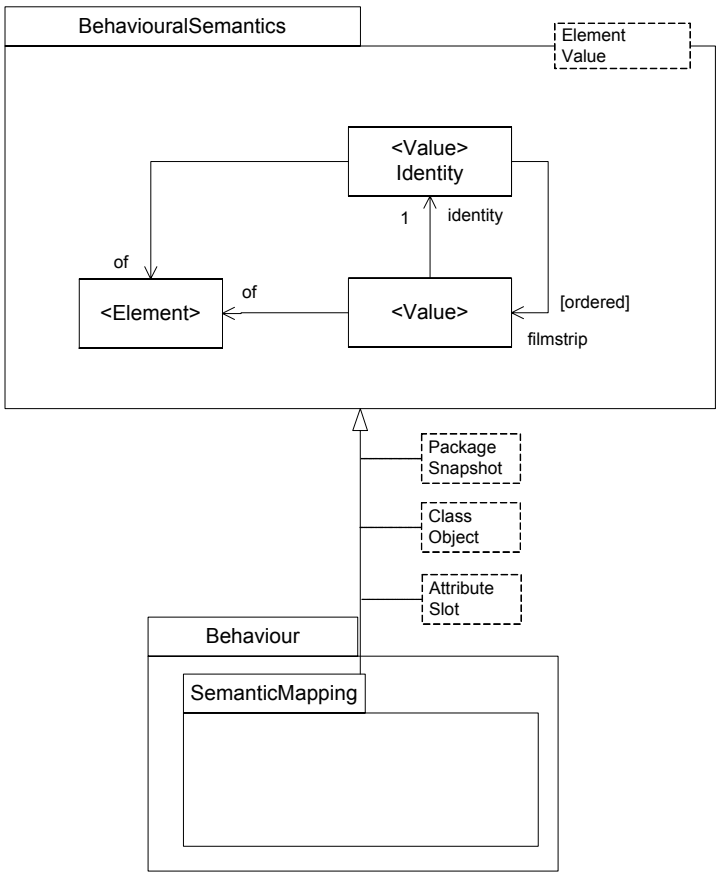


Figure 15-4 *Derivation of Behaviour Semantic Mapping package*

15.4.2 Model

Figure 15-5 on page 186 shows the Semantic Mapping for the Behaviour packages derived as illustrated in figure 15-4 on page 185. An instance of a package is an snapshot identity and a snapshot. The snapshot identity uniquely identifies a particular package instance and the snapshot describes the evolution of a particular package instance over time. An instance of a class is an object identity and a slot identity. An instance of an attribute is a slot identity and a slot.

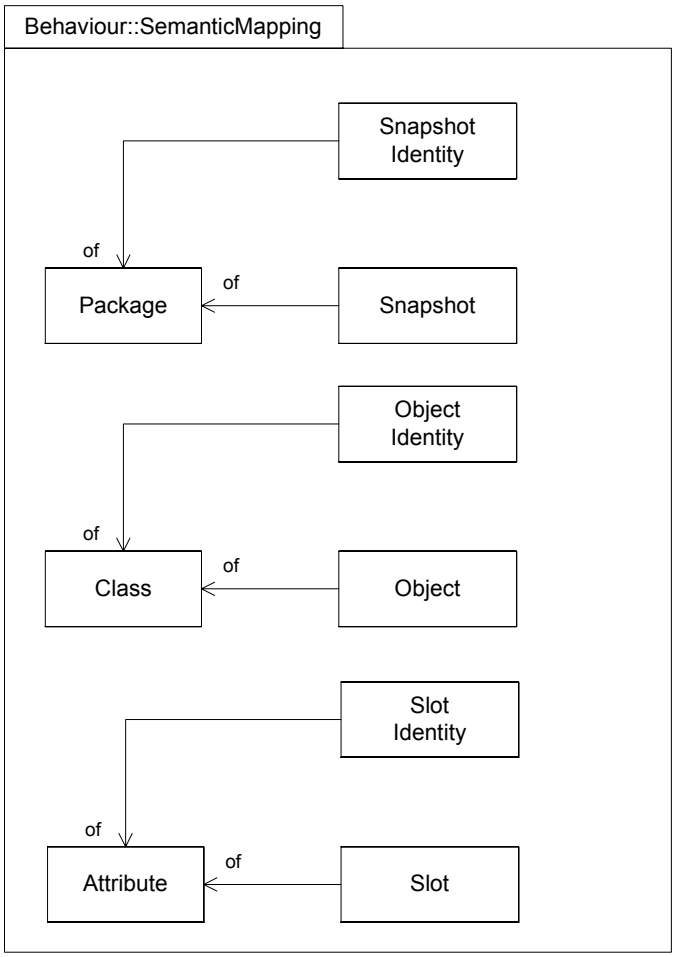


Figure 15-5 *Semantic Mapping for Behaviour package*

Snapshot

Associations

identity The identity of the snapshot.

of The package the snapshot is an instance of.

SnapshotIdentity

Associations

filmstrip An ordered set of snapshot.

of The package the snapshot identity is an instance of.

Object

Associations

identity The identity of the object.

of The class the object is an instance of.

ObjectIdentity

Associations

filmstrip An ordered set of objects.

of The class the object identity is an instance of.

Slot

Associations

identity The identity of the slot.

of The attribute the slot is an instance of.

SlotIdentity

Associations

filmstrip An ordered set of slots.

of The attribute the slot identity is an instance of.

15.4.3 Well-formedness Rules

Snapshot

[1] For each object owned by a snapshot there should be a class of the snapshot's package's namespace that the object is a value of.

```
context Snapshot inv:
 self.ownedObject->forAll(c |
   self.of.memberClass->exists(d | c.of = d))
```

[2] For each snapshot owned by a snapshot there should be a package of the snapshot's package's namespace that the snapshot is a value of.

```
context Snapshot inv:
 self.ownedSnapshot->forAll(c |
   self.of.memberPackage->exists(d | c.of = d))
```

Class

[1] For each slot owned by an object there should be an attribute of the object's class's namespace that the slot is a value of.

```
context Snapshot inv:
 self.ownedObject->forAll(c |
   self.of.memberClass->exists(d | c.of = d))
```

SnapshotIdentity

[1] All snapshots in the filmstrip should be of the same package as me.

```
context SnapshotIdentity inv:
  self.filmstrip->forAll(e1 | e1.of = of)
```

ObjectIdentity

[1] All objects in the filmstrip should be of the same class as me.

```
context ObjectIdentity inv:
    self.filmstrip->forAll(e1 | e1.of = of)
```

SlotIdentity

[1] All slots in the filmstrip should be of the same attribute as me.

```
context SnapshotIdentity inv:
    self.filmstrip->forAll(e1 | e1.of = of)
```

15.4.4 Operations

There are no operations

15.5 EXAMPLE SNAPSHOTS

Figure 15-7 on page 188 exemplifies how the definition introduced in this chapter enables the modelling of dynamic systems using the example shown in figure 15-6 on page 188.

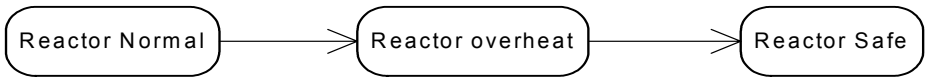


Figure 15-6 *Example of state changes*

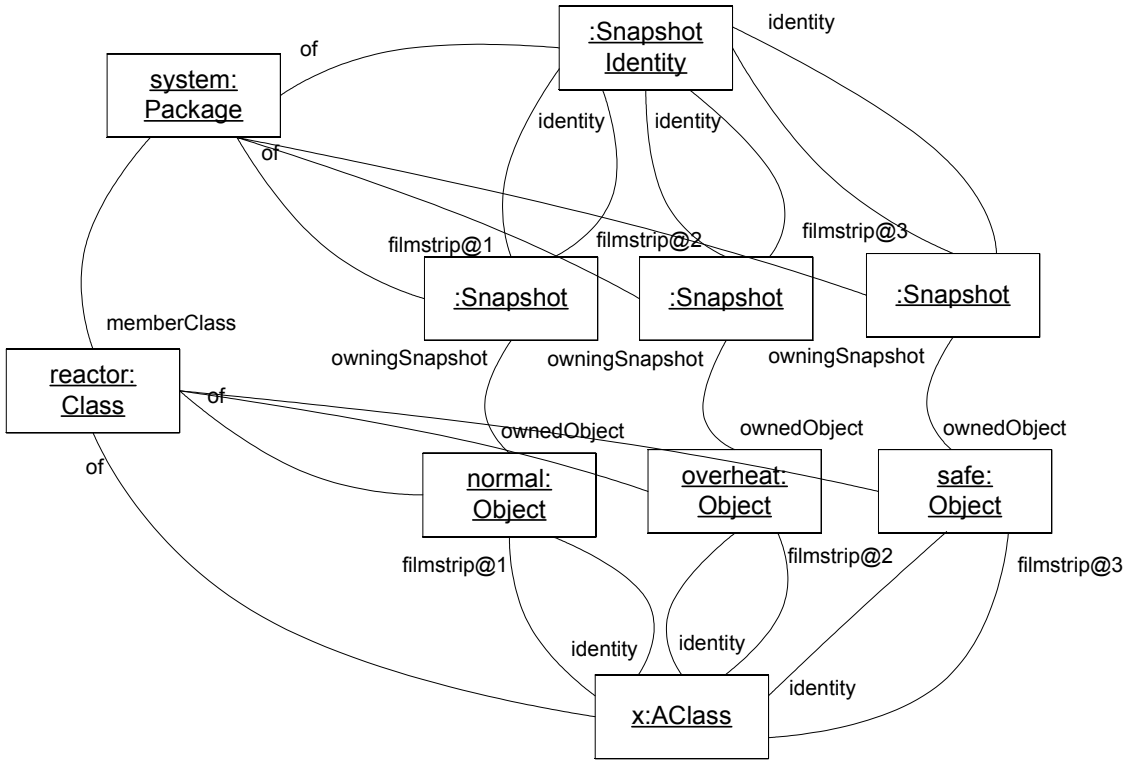


Figure 15-7 *Example snapshot of figure 15-5 on page 186*

This example models the evolution of an object through three states (normal, overheat and safe), the state change at the object level also forces a state change at the snapshot level. Collectively we can consider a model of a single state change as a time slice of the systems evolution. Time slices are related via the respective identities of model element instances.

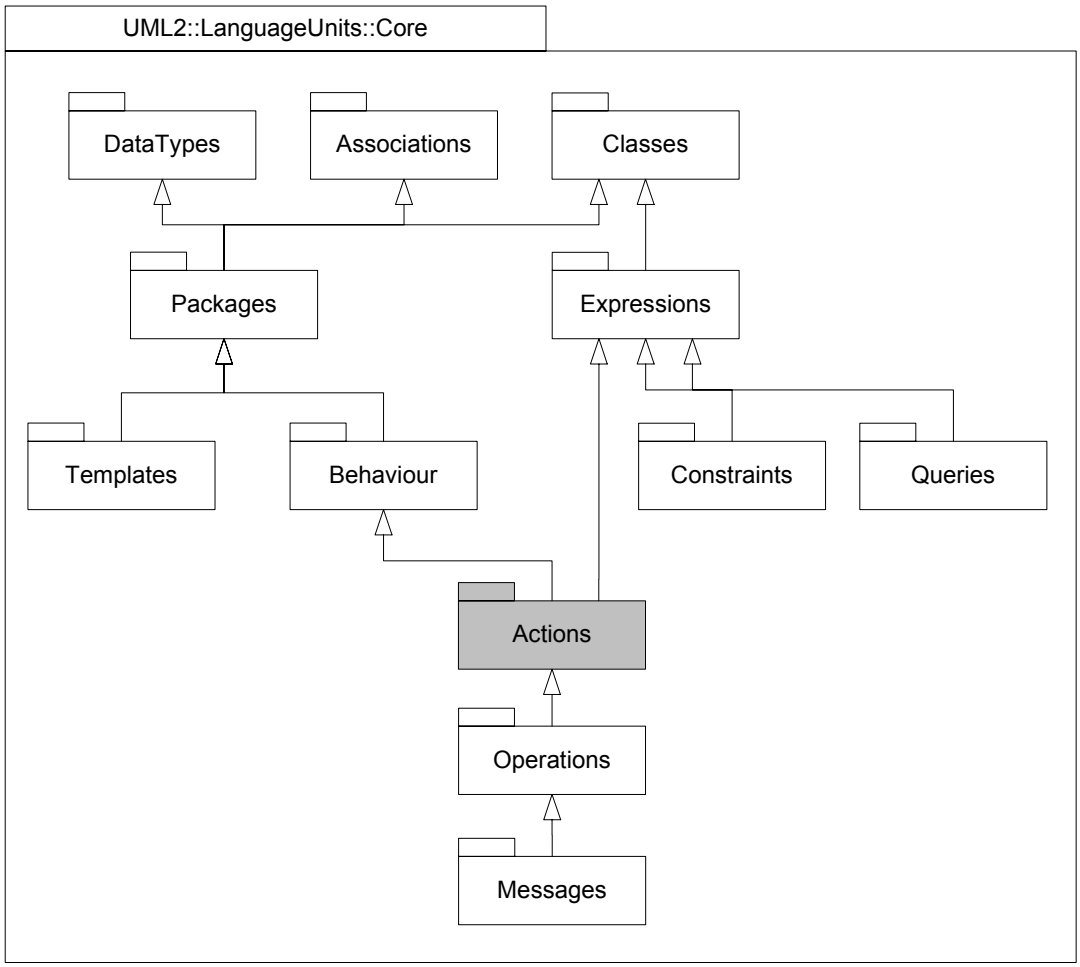
15.6 CHANGES TO UML 1.4

UML 1.4 does not have a model of behaviour. This chapter has provided a model that can be used as a foundation for understanding the semantics of UML's behavioural features.

Chapter 16
Actions

This package defines the abstract syntax and semantics of actions. Actions describe state changing computations of the system, and are used in the body of operations (chapter 17). This chapter is broadly split into two parts. The first part defines a small, but rich, action language. The second part defines the templates used to stamp out the action language.

16.1 POSITION IN ARCHITECTURE



The definition described in this chapter ultimately aims to precisely define a core subset of the actions described in the action semantic proposal [ActionSemantics]. To this end, it is not the intention to replace that proposal but to show how that definition can be derived using a template based approach. A characteristic of stamping out definitions using templates, is that some concepts can be left undefined while others are well defined. In the case of

the action definition presented here, the abstract syntax is non-normative and can be quickly substituted for any syntactical construct (therefore supporting families of action languages). The essence of the definition lies in its treatment of the semantic domain.

16.1.1 Example

Figure 16-1 on page 191 gives an example of a simple action that assigns the value 10 to the variable x.

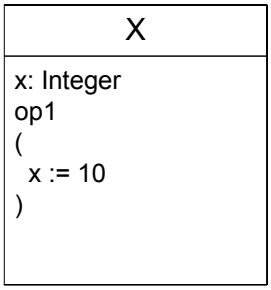


Figure 16-1 *Action example*

16.2 ABSTRACT SYNTAX

16.2.1 Derivation

Figure 16-2 on page 192 and figure 16-3 on page 193 show how the abstract syntax of the actions package is stamped out using the composite action abstract syntax template shown in figure 16-11 on page 206 for sequential and parallel actions, and the typed action operand abstract syntax template shown in figure 16-13 on page 208 for write attribute action and create object action.

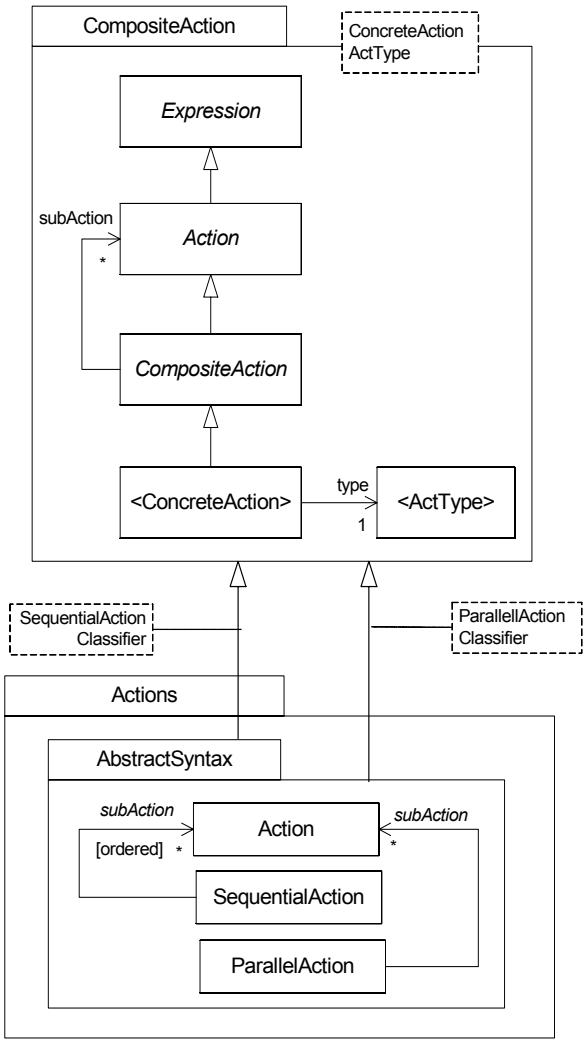


Figure 16-2 *Derivation of the abstract syntax for sequential and parallel actions*

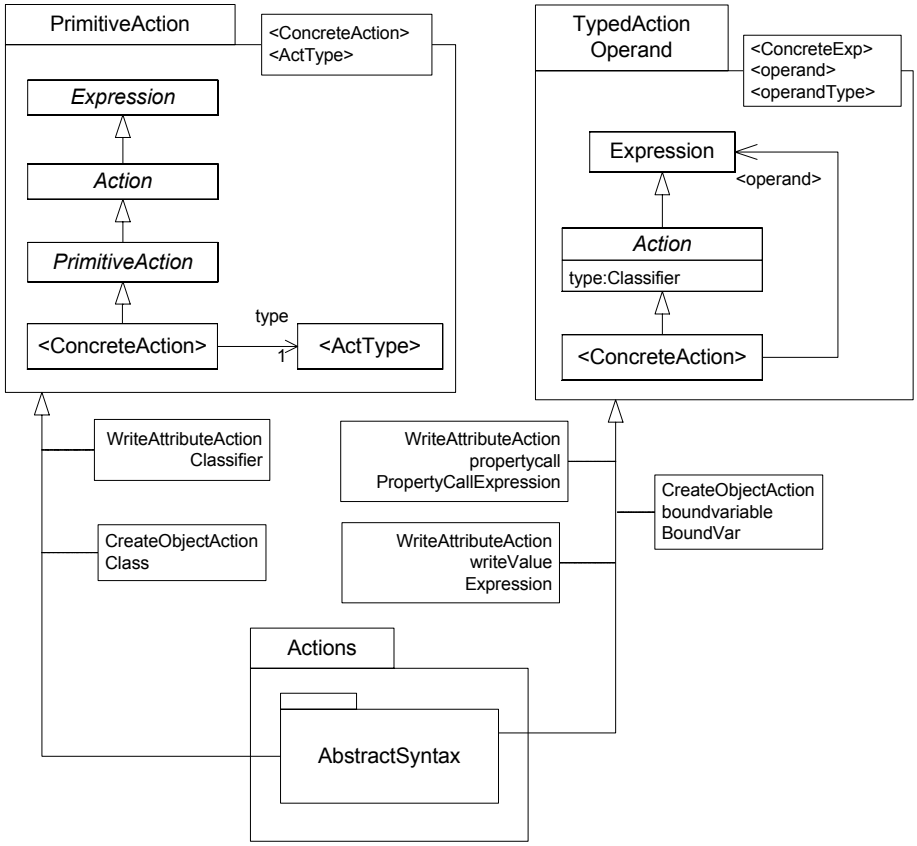


Figure 16-3 *Derivation of the abstract syntax for create object and write attribute actions*

16.2.2 Model

Figure 16-4 on page 194 shows the abstract syntax of the actions package. The action language consists of two primitive and two compound actions. The first primitive action is write attribute action which updates the value of an attribute. The two operands of a write attribute action are of type expression (expression is the superclass of both static expressions and actions) and can therefore be either further actions or static expressions, the first operand is constrained to be of type property call expression which must point to an attribute (see section 16.2.3 on page 195). The second primitive action is create object action which instantiates a class. The operand of a create object action can also be of type expression, however this is constrained to be a bound variable which binds a class (see section 16.2.3 on page 195). The first compound action is parallel action which has a number of sub actions (which can be either composite or primitive actions). The second compound action is sequential action which has a number of sub actions (which again can be either composite or primitive actions).

Expression

Expression is an abstract class purely used for the purposes of polymorphism. It is the plugin point for static expressions (see chapter 12) and therefore enables write attribute and create object actions to use static expressions or further actions as their operands.

Action

Action is an abstract class purely used for the purposes of polymorphism. It enables the type of concrete actions to be considered more generally as that of action.

CompositeAction

Composite action is an abstract class used purely for the purposes of polymorphism. It enables the type of sequential and parallel actions to be considered more generally as that of composite action.

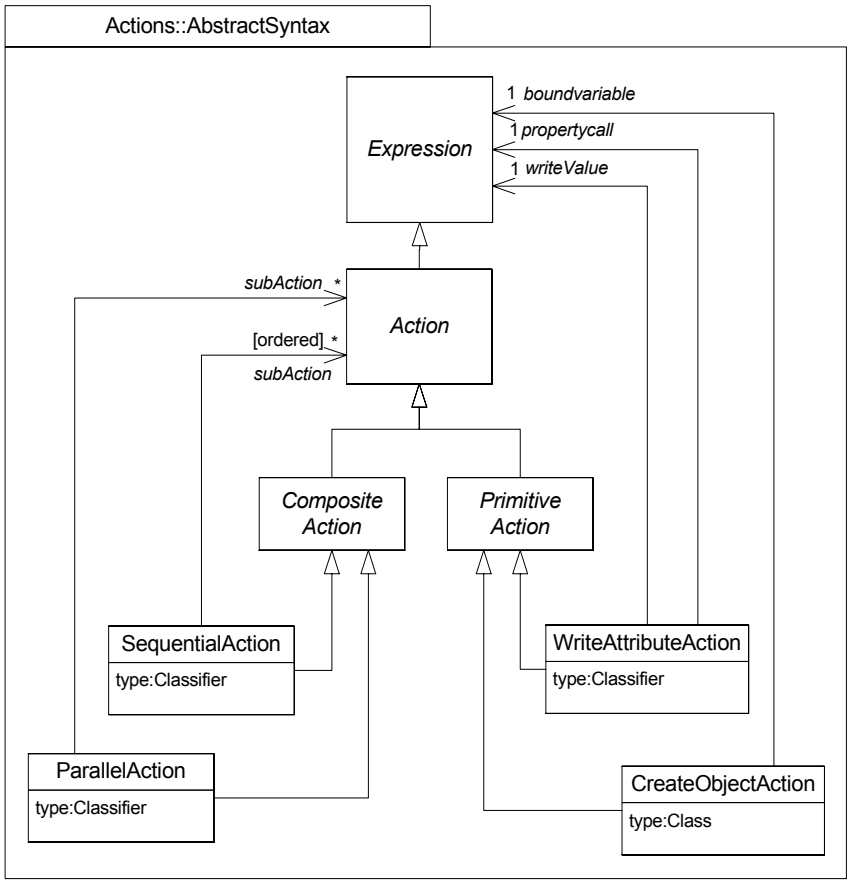


Figure 16-4 *Abstract syntax domain for Actions package*

PrimitiveAction

Primitive action is an abstract class used purely for the purposes of polymorphism. It enables the type of write attribute and create object actions to be considered more generally as that of primitive action.

ParallelAction

Parallel action is a concrete action which contains a set of sub actions. It is the syntax for a semantic domain entity which describes how sub actions should be executed in parallel.

Associations

type The type of the parallel action.

subActions A set of sub actions whose execution is controlled by the parallel action.

SequentialAction

Sequential action is a concrete action which contains an ordered set of sub actions. It is the syntax for a semantic domain entity which describes how the sub action are executed sequentially.

Associations

type The type of the sequential action.
subActions An ordered set of sub actions whose execution is controlled by the sequential action.

WriteAttributeAction

Write attribute action is a concrete action which describes the syntax for a semantic domain construct which updates the value of the left operand (propertyCall expression which refers to the attribute) with the value of the right operand.

Associations

type The type of the write attribute action.
propertycall The first operand of the write attribute action.
writeValue The second operand of the write attribute action.

CreateObjectAction

Create object action is a concrete action which describes the syntax for a semantic domain construct which creates an instance (object) of the class referenced by the action's operand.

Associations

type The type of the create object action.
boundvariable The operand of the create object action.

16.2.3 Well-formedness Rules

WriteAttributeAction

[1] The type of propertycall must be property call expression.

```
context WriteAttributeAction inv:
    self.propertycall.type.isKindOf(PropertyCallExpression)
```

[2] The type of writeValue must be expression.

```
context WriteAttributeAction inv:
    self.writeValue.type.isKindOf(Expression)
```

[3] The first operand of a write attribute action which is a property call expression must refer to an attribute.

```
context WriteAttributeAction
    self.propertycall.referedProperty.isKindOf(Attribute)
```

[4] propertycall's scope should include the scope of the containing write attribute action.

```
context WriteAttributeAction inv:
    self.scope->forAll(a | self.propertycall.scope->includes(a))
```

[5] writeValue's scope should include the scope of the containing write attribute action.

```
context WriteAttributeAction inv:
    self.scope->forAll(a | self.writeValue.scope->includes(a))
```

CreateObjectAction

[1] The operand of a create object action which is a bound variable must refer to a class.

```
context CreateObjectAction
    self.boundvariable.type.isKindOf(Class)
```

[2] boundvariable's scope should include the scope of the containing create object action.

```
context WriteAttributeAction inv:
    self.scope->forAll(a | self.boundvariable.scope->includes(a))
```

16.2.4 Operations

There are no operations.

16.3 SEMANTIC DOMAIN

16.3.1 Derivation

Figure 16-5 on page 197 and figure 16-6 on page 198 show how the semantic domain of the actions package is stamped out using the composite action evaluation semantic domain template for sequential and parallel action evaluations, the primitive action evaluation semantic domain template and the action operand evaluation semantic domain template for write attribute action evaluation and create object action evaluation.

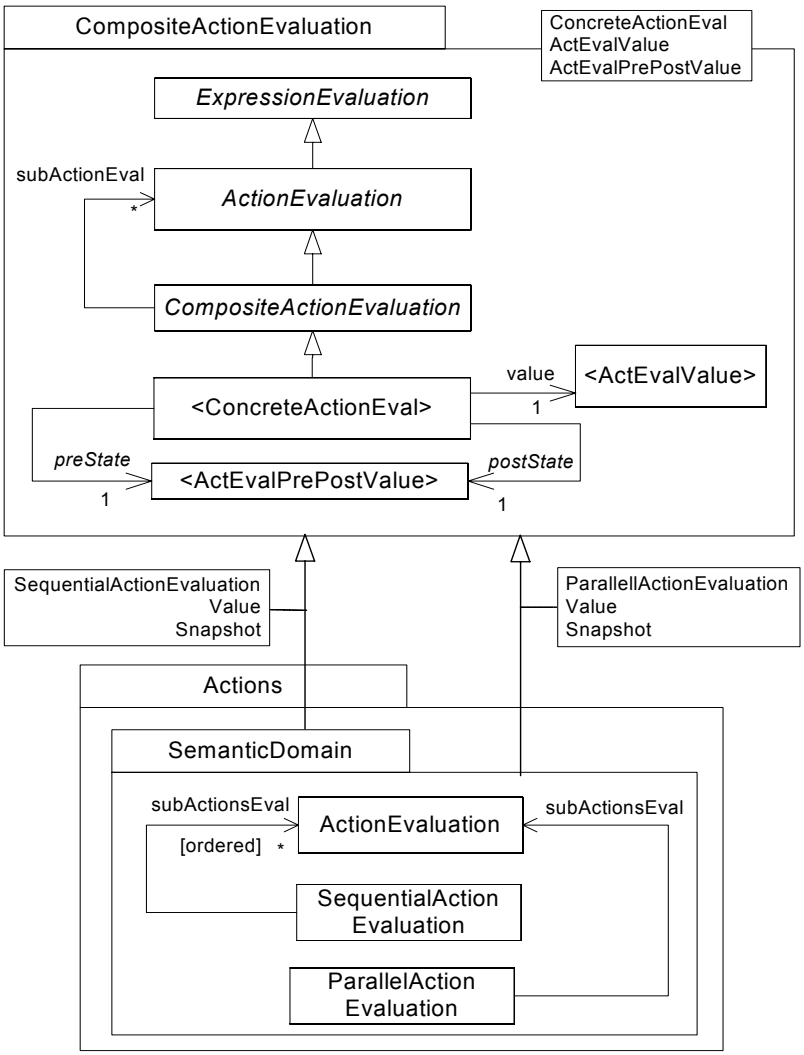


Figure 16-5 *Derivation of semantic domain for sequential and parallel actions*

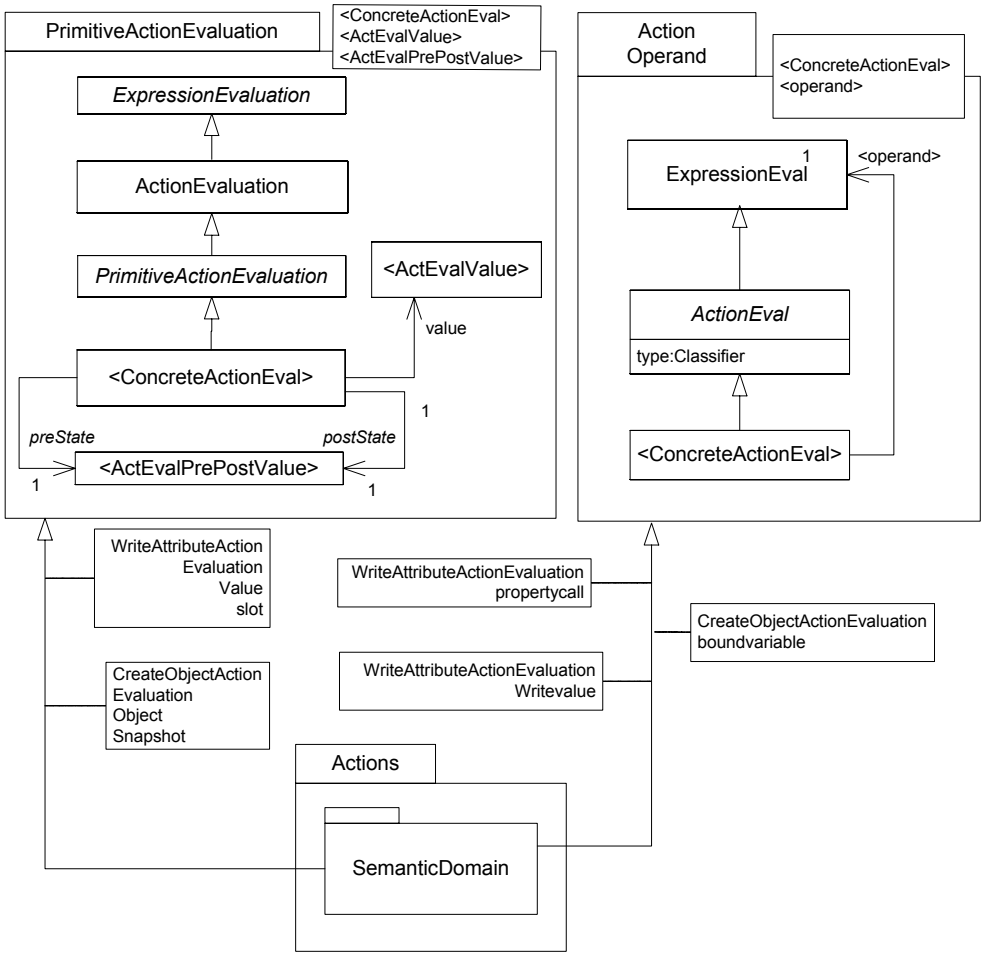


Figure 16-6 *Derivation of semantic domain for write attribute and create object actions*

16.3.2 Model

Figure 16-7 on page 199 shows the semantic domain of the actions package. This definition describes how each of the abstractions within the abstract syntax (described in section 16.2 on page 191) has a semantic domain evaluation. Each of the four concrete action evaluation (parallel, sequential, write attribute and create object evaluation action) have a pre and post state which capture the state of the system before and after the action has executed. The concrete action evaluations also have a value which they evaluate to upon execution.

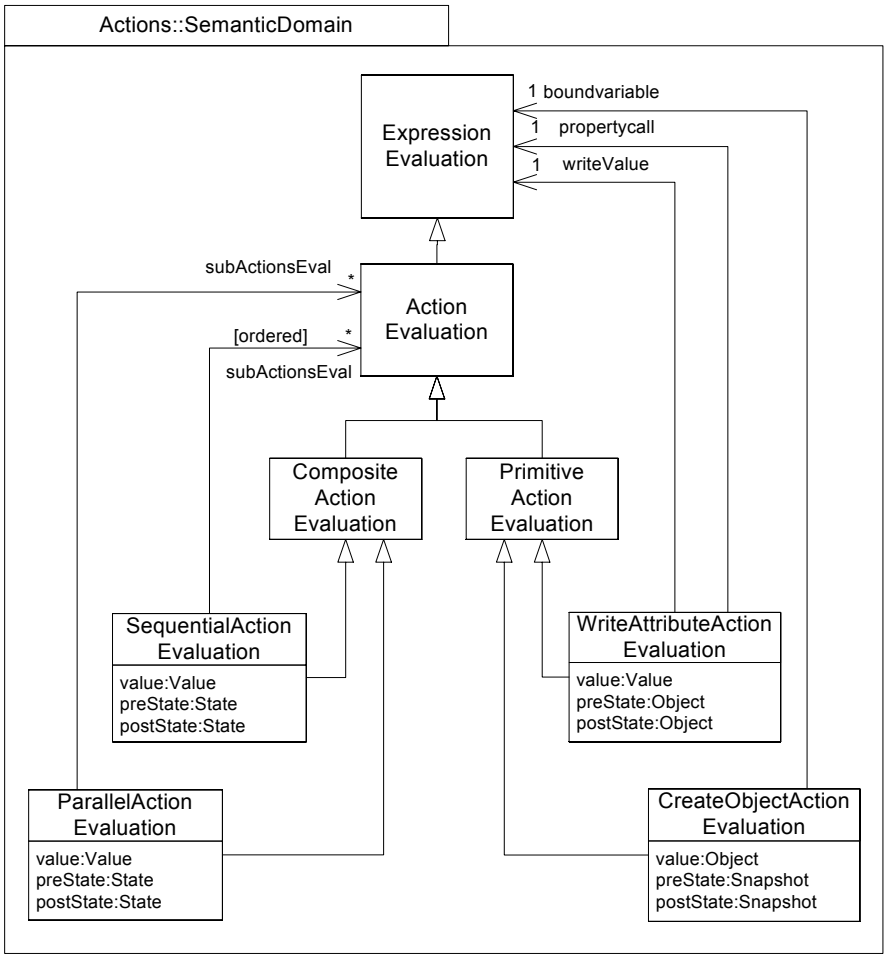


Figure 16-7 *Semantic domain for Actions package*

ExpressionEvaluation

Expression evaluation is an abstract class purely used for the purposes of polymorphism. It is the plugin point for static expression evaluation (see chapter 12) and therefore enables write attribute action evaluations and create object action evaluations to use static expressions evaluations or further action evaluations as their operand evaluations.

ActionEvaluation

Action evaluation is an abstract class purely used for the purposes of polymorphism. It enables the type of concrete action evaluations to be considered more generally as that of action evaluation.

CompositeActionEvaluation

Composite action evaluation is an abstract class used purely for the purposes of polymorphism. It enables the type of sequential and parallel action evaluations to be considered more generally as that of composite action evaluation.

PrimitiveActionEvaluation

Primitive action evaluation is an abstract class used purely for the purposes of polymorphism. It enables the type of write attribute action evaluation and create object action evaluation to be considered more generally as that of primitive action evaluation.

ParallelActionEvaluation

Parallel action is a concrete action evaluation which contains a set of sub action evaluations. It describes how sub action evaluations can be executed in parallel.

Associations

value The value of the parallel action evaluation.
subActionsEval A set of sub action evaluations whose evaluation is controlled by the parallel action.
preState A snapshot of the system before the parallel action evaluation executes.
postState A snapshot of the system after the parallel action evaluation executes.

SequentialActionEvaluation

Sequential action evaluation is a concrete action evaluation which contains an ordered set of sub action evaluations. It describes how sub action evaluations can be executed sequentially.

Associations

value The value of the sequential action evaluation.
subActionsEval Set of sub action evaluations whose evaluation is controlled by the sequential action.
preState A snapshot of the system before the sequential action evaluation executes.
postState A snapshot of the system after the sequential action evaluation executes.

WriteAttributeActionEvaluation

Write attribute action evaluation is a concrete action evaluation which describes how an attribute instance (slot) is updated with a value.

Associations

value The value of the write attribute action evaluation.
propertycall The slot (attribute instance) to update.
writeValue The value to update the slot with.
preState A snapshot of the system before the write attribute action evaluation executes.
postState A snapshot of the system after the write attribute action evaluation executes.

CreateObjectActionEvaluation

Create object action evaluation is a concrete action evaluation which describes how a new object is created.

Associations

value The value of the write attribute action evaluation.
boundvariable A bound variable instance (note: this is redundant but mirrors abstract syntax)
preState A snapshot of the system before the create object action evaluation executes.
postState A snapshot of the system after the create object action evaluation executes.

16.3.3 Well-formedness Rules

ParallelActionEvaluation

[1] The pre state of at least one subAction is at the same time slice (state and time slice is defined in chapter 15) as the pre state of the parallel Action.

```
context ParallelActionEvaluation
  not self.subActionsEval->forAll(a | not a.preState.isSameTime(self.preState))
```

[2] The post state of at least one subAction is at the same time slice as the post state of the parallel Action.

```
context ParallelActionEvaluation
  not self.subActionsEval->forAll(a|not a.postState.isSameTime(self.postState))
```

[3] The pre and post states of subActions lie between the pre and post state of the parallel Action.

```
context ParallelActionEvaluation
  self.subActionsEval->forAll(( a|a.preState.isSameTime(self.preState) or
     a.preState.isLater(self.preState)) and
     ( a.postState.isSameTime(self.postState) or
        a.postState.isEarlier(self.postState)) )
```

SequentialActionEvaluation

[1] All sub action evaluations should execute in sequence.

```
context SequentialActionEvaluation
  self.subActionsEval.zip(self.subAction.tail)->forAll(pair |
    pair->at(1).preState.isLater(pair->at(0).postState))
```

[2] The pre state of the first subAction is at the same time slice as the pre state of the sequentialAction.

```
context SequentialActionEvaluation
  self.preState.isSameTime(self.subActionsEval->at(0).preState)
```

[3] The post state of the last subAction is at the same time slice as the post state of the sequentialAction.

```
context SequentialActionEvaluation
  self.postState.isSameTime(self.subActionsEval->last().postState)
```

WriteAttributeActionEvaluation

[1] The pre and the post state of the write attribute action evaluation must be the same instance.

```
context WriteAttributeActionEvaluation
  self.preState.identity = self.postState.identity
```

[1] The slot referred to in the propertycall must be owned by the Object in the pre state.

```
context WriteAttributeActionEvaluation
  self.preState.ownedSlot->includes(self.propertycall.referedProperty)
```

[2] An attribute evaluation results in updating the slot of the object in pre state with the value of the second operand.

```
context WriteAttributeActionEvaluation
  self.postState.ownedSlot->iterate(i s=Set{} |
```

```
      if i.identity = self.propertycall.referedProperty.identity
    then
        s->union({i})
    else
        s)->forAll(s | s.value = self.writeValue.value)
```

[3] The value of the write attribute action is the value of the second operand.

```
context WriteAttributeActionEvaluation
  self.value = self.writeValue.value
```

CreateObjectActionEvaluation

[1] A create object evaluation results in the existence of an object in the post state that did not exist in the pre state.

```
context CreateObjectActionEvaluation
  self.preState.ownedObject->symmetricDifference(
    self.postState.ownedObject)->size = 1 and
      self.preState.ownedObject->size = self.postState.ownedObject-> size()-1
```

[2] The value of a create object action evaluation is the new object created.

```
context CreateObjectActionEvaluation
  self.value = self.preState.ownedObject->symmetricDifference
    (self.postState.ownedObject)->asSequence()->at(0)
```

[3] A unique id for the new object must be created.

```
context CreateObjectActionEvaluation inv:
  self.value.id.filmstrip->size() = 1
```

16.3.4 Operations

There are no operations.

16.4 SEMANTIC MAPPING

16.4.1 Derivation

The derivation of the semantic mapping of actions is shown in figure 16-8 on page 203. This illustrates how four stampings of the semantics template are used to form the derivation.

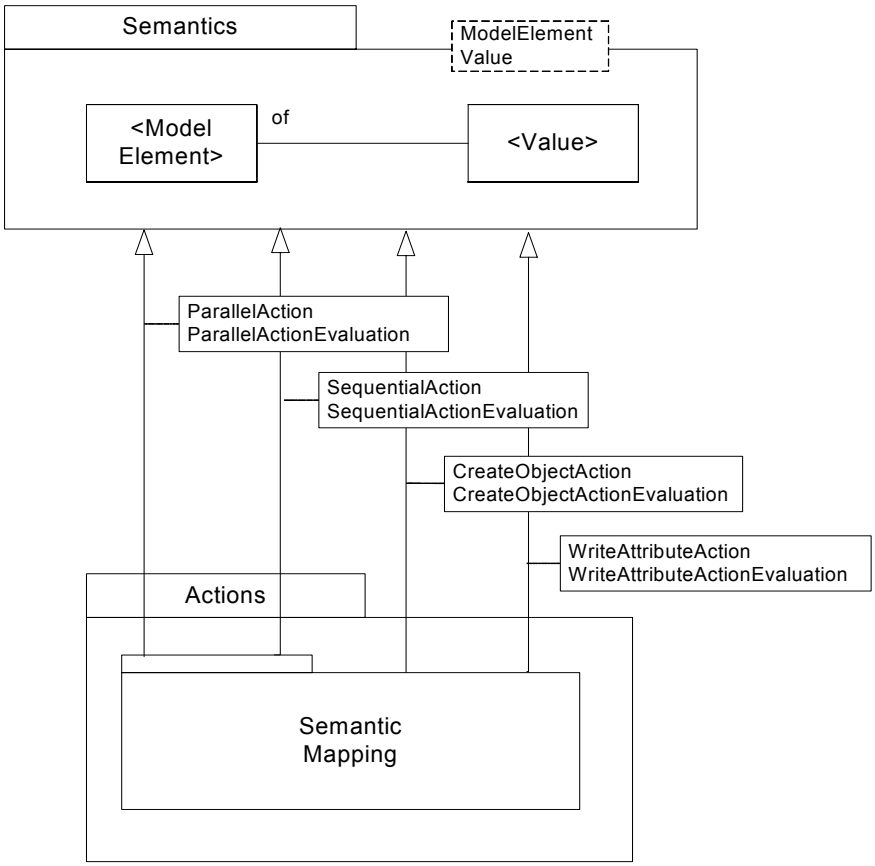


Figure 16-8 *Derivation of the Actions semantic mapping package*

16.4.2 Model

The semantic mapping of the actions package is shown in figure 16-9 on page 203. This describes how actions have evaluations.

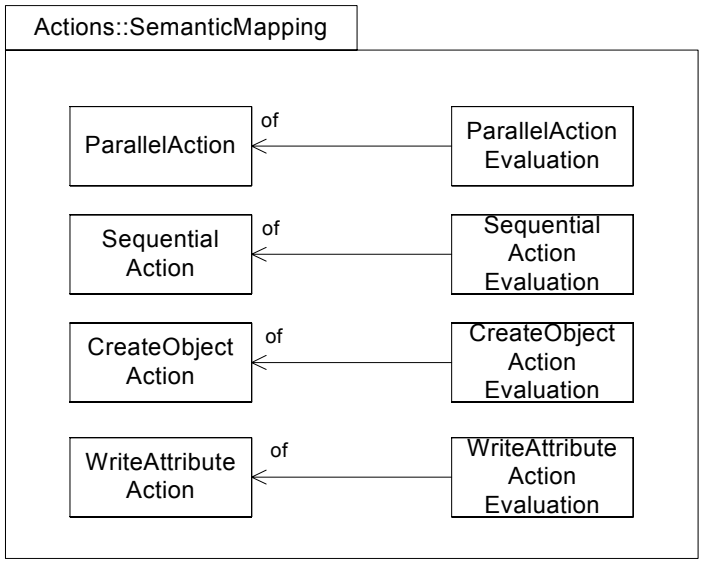


Figure 16-9 *Semantic mapping for the Actions package*

ParallelActionEvaluation

Associations

of The parallel action that the parallel action evaluation is an instance of.

SequentialActionEvaluation

Associations

of The sequential action that the sequential action evaluation is an instance of.

CreateObjectEvaluation

Associations

of The create object action that the create object action evaluation is an instance of.

WriteAttributeEvaluation

Associations

of The write attribute action that write attribute action evaluation is an instance of.

16.4.3 Well-formedness Rules

WriteAttributeActionEvaluation

[1] The propertycall value must conform to the operand type.

```
context WriteAttributeActionEvaluation inv:
     self.propertycall.value.of.conformsTo(self.of.propertycall.type)
```

[2] The writeValue value must conform to the operand type.

```
context WriteAttributeActionEvaluation inv:
     self.writeValue.value.of.conformsTo(self.of.writeValue.type)
```

CreateObjectActionEvaluation

[1] The new object created must be of the type of the bound variable referenced in the actions syntactical operand.

```
context createObjectActionEvaluation
  self.preState.ownedObject->symmetricDifference(
    self.postState.ownedObject->forAll(obj | obj.type =
  self.boundvariable.value)
```

[2] The boundVar value must conform to the boundvariable's type.

```
context CreateObjectActionEvaluation inv:
     self.boundvariable.value.of.conformsTo(self.of.boundvariable.type)
```

16.4.4 Operations

There are no operations.

16.5 SNAPSHOT

Figure 1-1 on page 157 shows a snapshot of the evolution of the write attribute action shown in figure 16-1 on page 191. Note that there is only a partial mapping between the elements of abstract syntax and semantic domain for brevity of presentation. In this snapshot a write attribute action is contained by an operation (operations are dealt with in detail in chapter 17). Prior to the write attribute action (in its preState) the value of the slot, corresponding to the attribute x, is 5. After the write attribute action has evaluated (its post state) the slot is bound with the value 10.

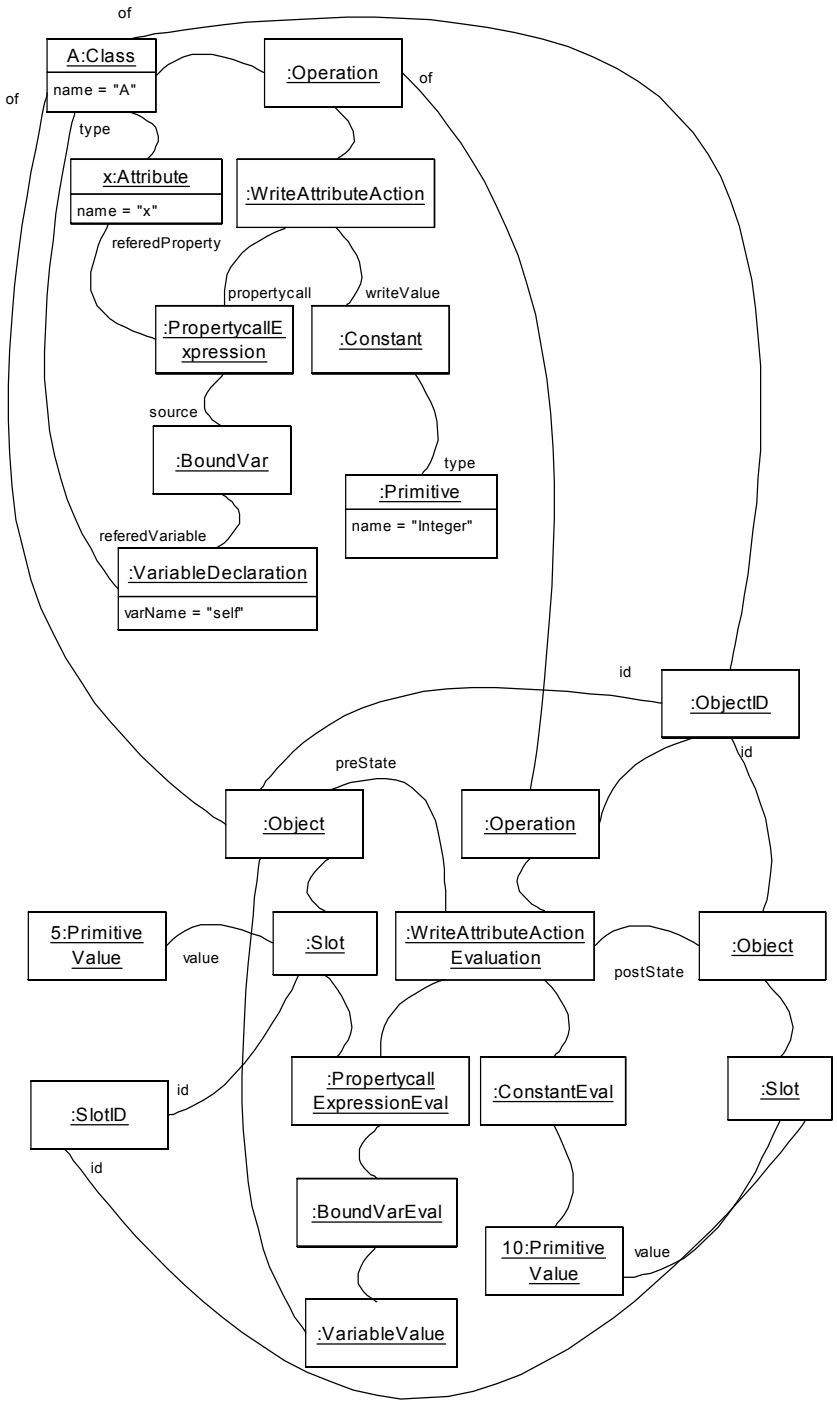


Figure 16-10 *Snapshot of write attribute action*

16.6 CHANGES TO UML 1.4

The submission defines the semantics of two key action concepts in UML 1.4: object creation and send actions (see chapter 18 for the latter) and three key concepts from the action semantics submission: sequential, parallel actions and write actions.

16.7 TEMPLATES

This section describes a set of templates which capture the essence of actions and are generic enough to stamp out a family of action languages.

16.7.1 Primitive and compound action

Primitive and compound action templates are the basic building blocks for the action definition presented in this chapter. The role of these two templates is to classify actions as either primitive or compound. Primitive actions have no sub actions whereas compound actions have a set of sub actions.

Templates

Figure 16-11 on page 206 shows the abstract syntax templates for primitive and composite actions. A concrete primitive action is a generalized primitive action. A concrete composite action is a generalized composite action.

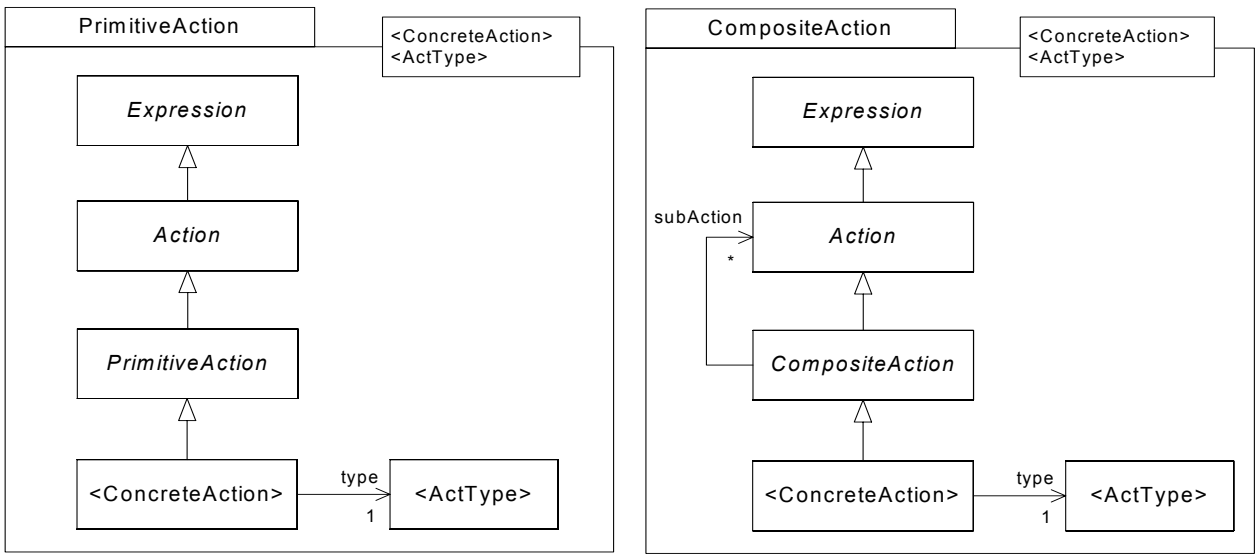


Figure 16-11 *Primitive and composite actions abstract syntax templates*

Actions extend expression and hence have scope. A definition of expression and scope is given in chapter 12.

Within the typed Composite Action template, The subactions of the composite action template must include in its scope the scope of the composite action.

```
context CompositeAction inv:
  self.subAction->forAll( subScope |
    self.scope->forAll(selfScope | subScope->includes(selfScope)))
```

Figure 16-12 on page 207 shows the semantic domain templates for primitive and composite action evaluations. A concrete primitive action evaluation is a generalized primitive action evaluation and has a pre and post state describing its evaluation. A concrete compound action evaluation is a generalized compound action evaluation and also has a pre and post state describing its evaluation.

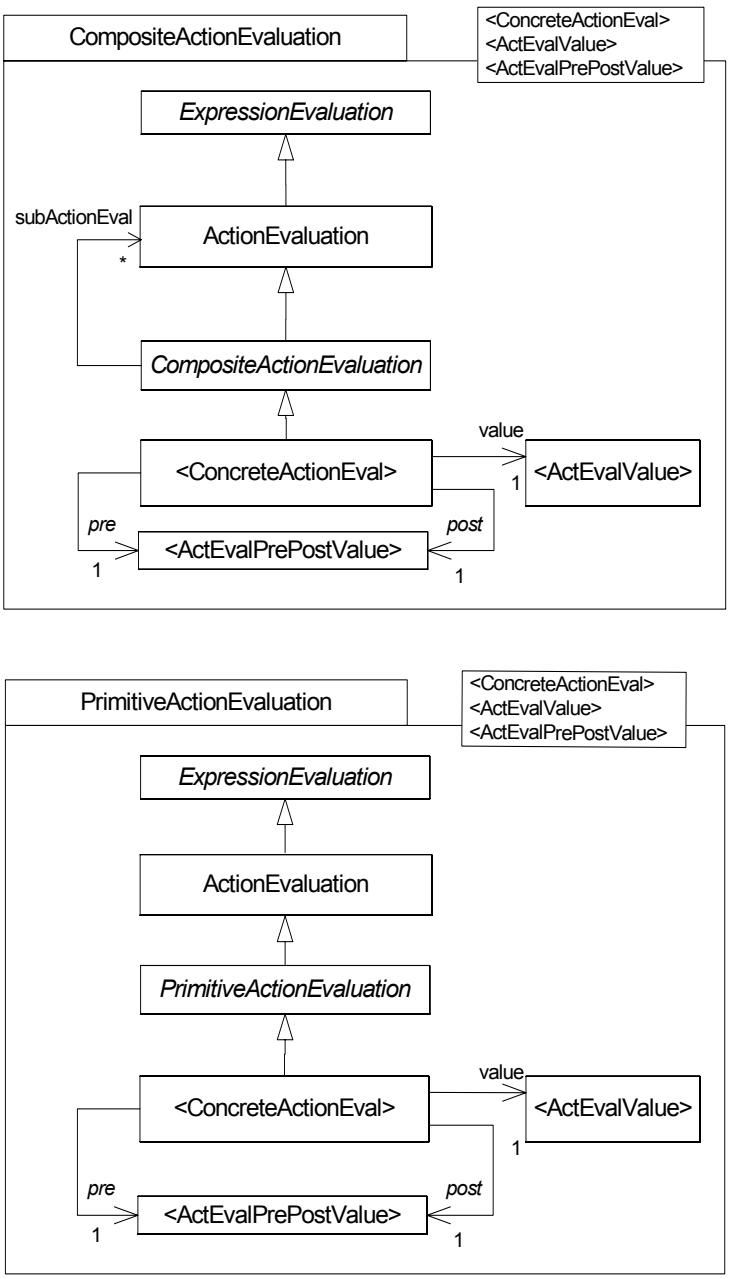


Figure 16-12 *Primitive and composite action semantic domain templates*

16.7.2 Action Operands

Primitive actions have operands that are of type expression which means they can contain further actions or static expressions (because static expressions generalize expression, see chapter 12). In this section we describe templates that add operands to actions.

Templates

Figure 16-13 on page 208 shows the two templates for adding operands to actions. The first template (ActionOperand) is a basic operand template, which adds to an Action a single operand, which is an expression. The second template (TypedActionOperand) augments the first template by adding a constraint on the return type of the operand and hence has an additional parameter operand type.

It should be noted that semantic domain and semantic mapping templates for typed Action operands are not required, since expression values are already checked against type in the ActionOperandMap template (see below).These template (and the corresponding semantic domain and semantic mapping templates) can be stamped out multiple times for multiple operands.

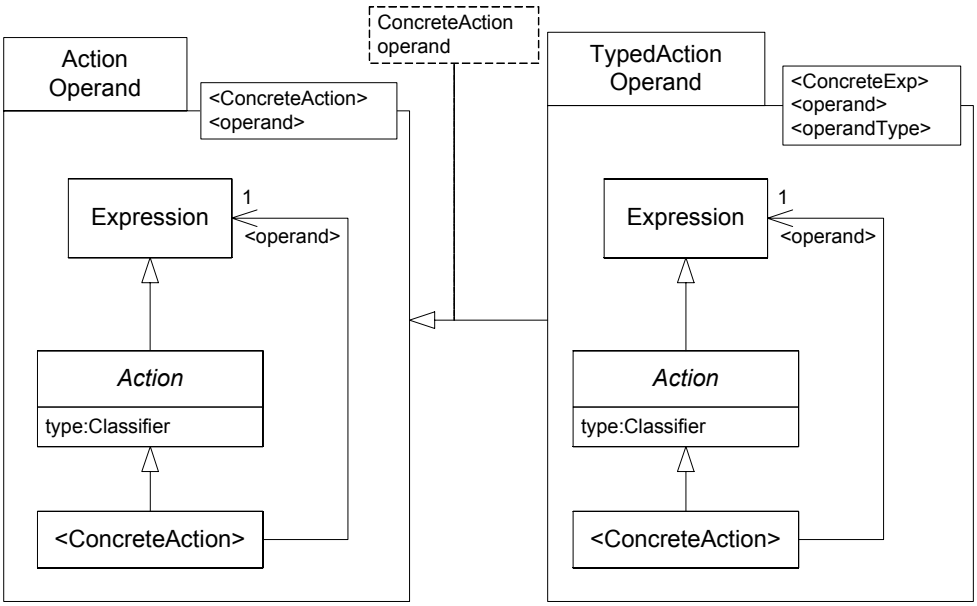


Figure 16-13 *Abstract syntax template for adding operands to actions*

Within the typed Action operand template, an operand's scope should include the scope of the containing action.

```
context <ConcreteAction> inv:
  self.scope->forAll(a | self.<operand>.scope->includes(a))
```

Also within the typed Action template, an operand's type should match the type specified in the parameters. This is expressed using the following constraint:

```
context <ConcreteAction> inv:
  self.<operand>.type.isKindOf(<operandType>)
```

Figure 16-14 on page 209 shows the semantic domain templates for Action operands. An action evaluation has an operand, which is a expression evaluation.

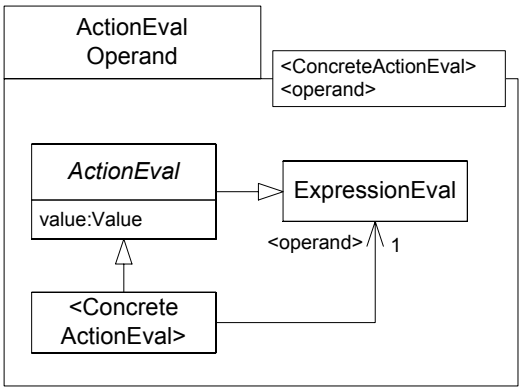


Figure 16-14 *Semantic Domain Template for Action Operands*

Figure 16-15 on page 209 shows the semantic mapping templates for static expression operands.

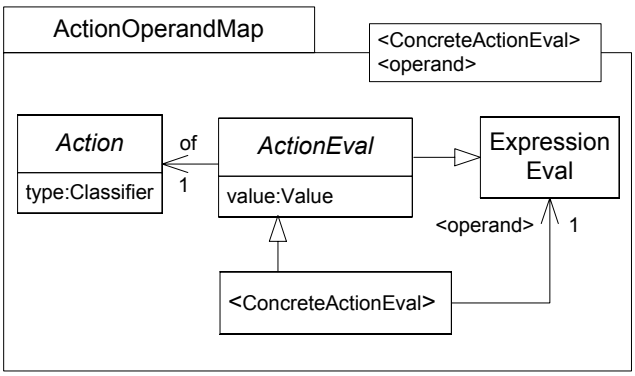


Figure 16-15 *Semantic Mapping Templates for Action Operands*

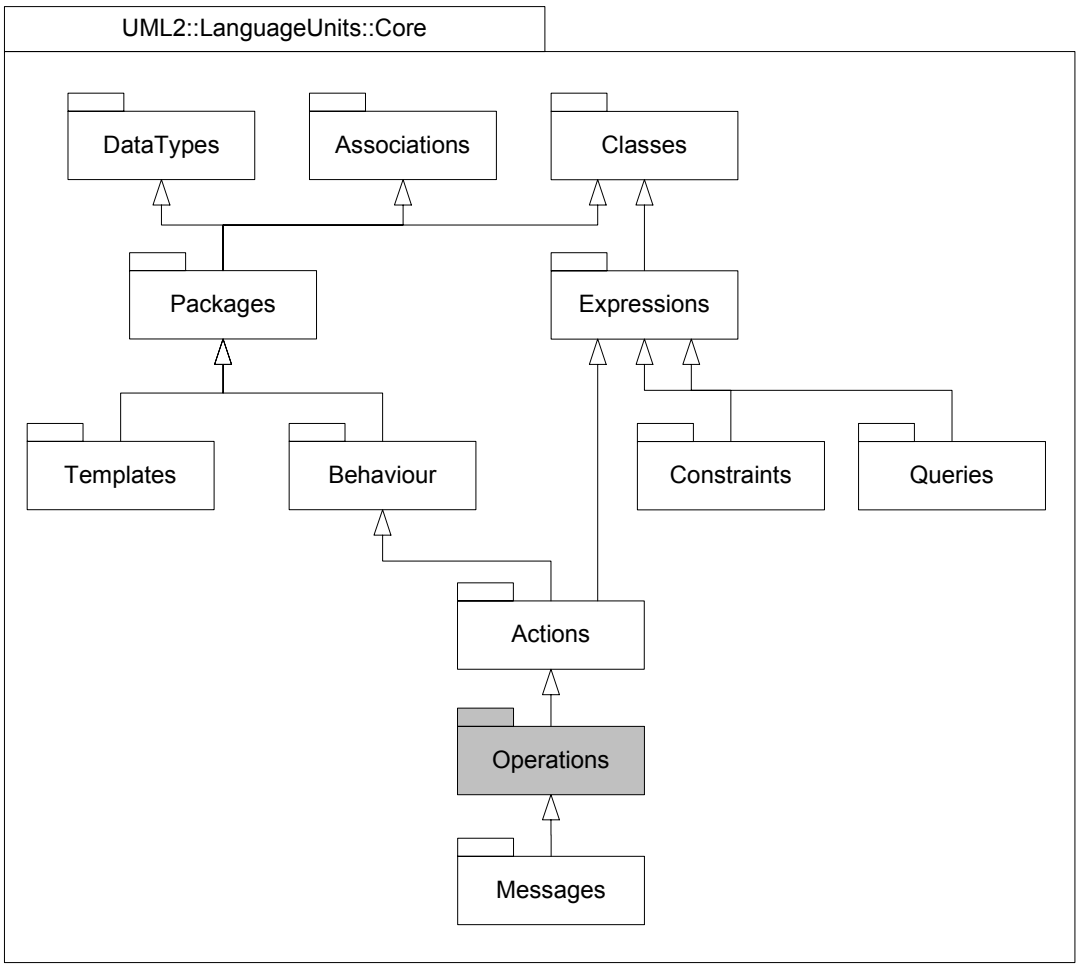
An Primitive Action's operand evaluations should be valid in view of its type. This is expressed using the following constraint:

```
context <ConcreteActionEval> inv:
  self.<operand>.value.of.conformsTo(self.of.<operand>.type)
```

Chapter 17
Operations

This chapter describes the definition of operations. Operations faclitate the abstract specification of state changes through their pre- and post-conditions. Operations may also reference actions (see chapter 16) thus supporting the refinement of abstract specifications of behaviour into executable action expressions.

17.1 POSITION IN ARCHITECTURE



17.1.1 Example

Figure 17-1 on page 211 shows an example of a simple operation, incr(), that increments the variable y provided that its value is zero.

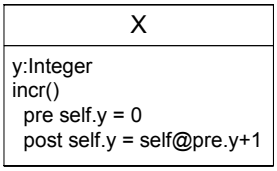


Figure 17-1 *Example operation*

17.1.2 Derivation

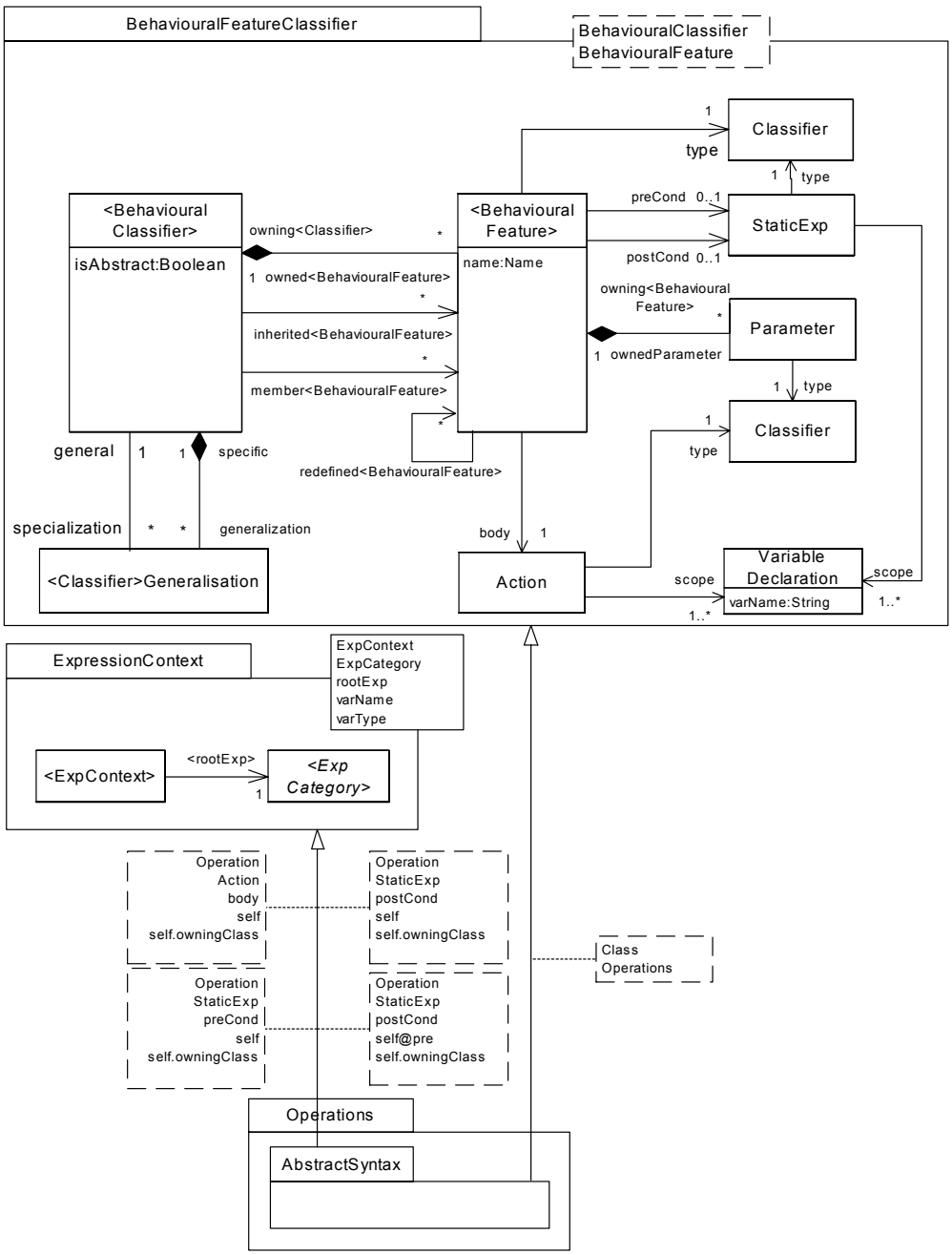


Figure 17-2 *Derivation of Operations abstract syntax package*

17.1.3 Model

Figure 17-3 on page 212 shows the abstract syntax for the operations package derived as illustrated in figure 17-2 on page 211. An operation is contained by a class and has a type, an operation can also have zero or many parameters and may have a pre and post condition. The pre and post conditions constrain the state of the system before and after the execution of the operation. The body of an operation evaluation is described by an action which is the root of an action tree.

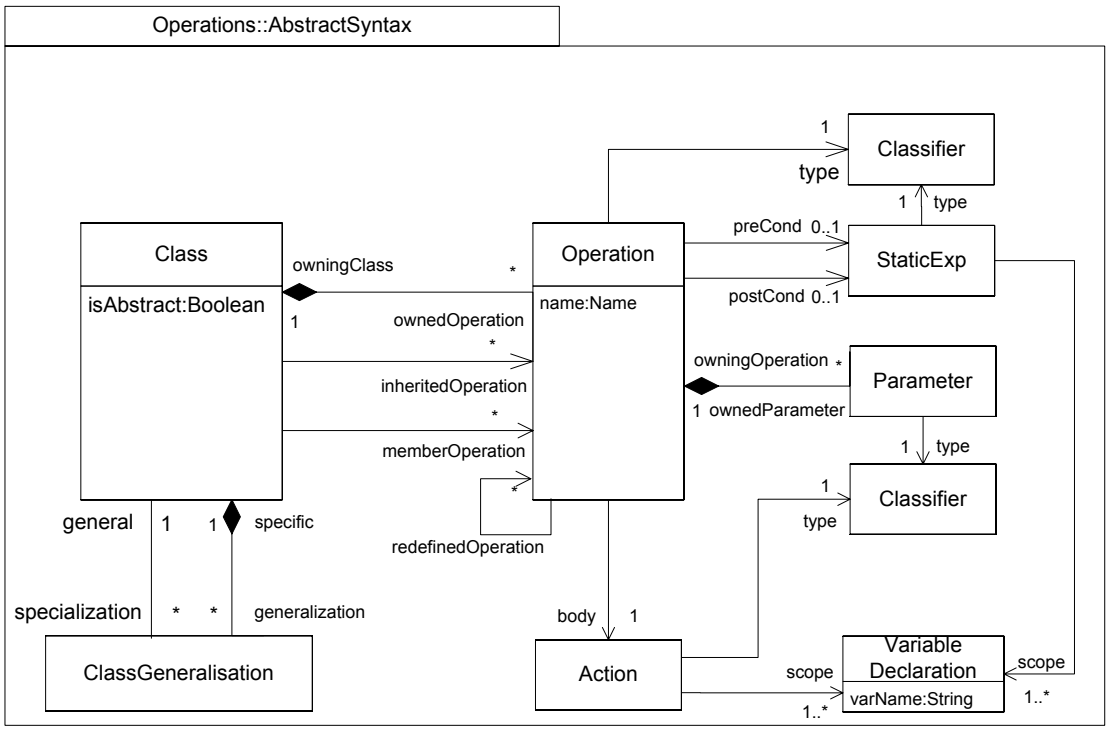


Figure 17-3 *Abstract syntax for Operations package*

Class

Attributes

isAbstract True if the class is abstract

Associations

inheritedOperation The inherited operations of the classifier.

memberOperation The operations that are members of the namespace of the class.

ownedOperation The operations owned by the classifier.

specialization The specializations of the class.

generalization The generalizations of the class.

Operation

Associations

body The body of the operation.

name The name of the operation.

owningClass The class that owns/contains the operation.

ownedParameter The parameters of the operation.

preCond The pre condition of the operation.

postCond The post condition of the operation.

redefinedOperations The operations that are redefined.

type The type of the operation.

ClassGeneralization

Associations

general The general class.

specific The specific class.

StaticExp

Associations

type The type of the static expression.

Parameter

Associations

owningOperation The operation that owns/contains the parameter.

type The type of the parameter.

Action

Associations

type The type of the action.

17.1.4 Well-formedness Rules

Class

[1] The members of a class must include the owned operations of the class.

```
context Class inv:
  self.memberOperation->includesAll(self.ownedOperation)
```

[2] Circular inheritance is not permitted.

```
context Class inv:
  not self.allGeneralElements()->includes(self)
```

[3] Parent element's operations must be inherited.

```
context Class inv:
  self.inheritedOperation = self.generalElements()->iterate(p s = Set{} |
    s->union(p.memberOperation->reject(x |
      self.memberOperation->exists(x'|
        x'.redefinedOperation->includes(x)))))
```

[4] Member operations must include the inherited features.

```
context Class inv:
  self.memberOperation->includesAll(self.inheritedOperation)
```

[5] Member operations may only redefine parent features.

```
context Class inv:
  self.memberOperation->forAll(x |
    (self.generalElements() -> iterate(s = Set{} |
      s->union(g.memberOperation))))->includesAll( x.redefinedOperation)
```

Operation

[1] Redefined operations must conform.

```
context Operation inv:
  self.redefinedOperation->forAll(f |
    self.type.conformsTo(f.type))
```

[2] The pre and post condition expressions of an operation must be of type boolean.

```
context Operation inv:
  self.preCond.type = boolean and self.postCond.type = boolean
```

[3] The scope of the operation's action must include self.

```
context Operation inv:
  self.body.scope->exists(v | v.varName = self
    and v.type = self.owningClass)
```

[4] The scope of the operation's pre condition must include self.

```
context Operation inv:
  self.preCond.scope->exists(v | v.varName = self
    and v.type = self.owningClass)
```

[5] The scope of the operation's post condition must include self.

```
context Operation inv:
  self.postCond.scope->exists(v | v.varName = self
    and v.type = self.owningClass)
```

[6] The scope of the operation's post condition must include self@pre.

```
context Operation inv:
  self.postCond.scope->exists(v | v.varName = self@pre
    and v.type = self.owningClass)
```

17.1.5 Operations

Class

[1] Looks up a operation in a class given a name.

```
context Class::lookupOperationforName(x:Name):featureClassifier::
  Operation
  self.memberOperation->select(e|e.name = x ).selectElement()
```

[2] Looks up a name in a class given a operation.

```
context Class::lookupNameForOperation(x : Operation): Name
    self.memberOperation->select(e|e = x ).selectElement().name
```

[3] Returns the generalizations of the class.

```
context Class::generalElements() : Set(Class)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[4] Transitively returns all generalizations of the class.

```
context Class::allGeneralElements(): Set(Class)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

Operation

[1] Checks whether the supplied operation is in the same class as the operation.

```
context Operation::sameNamespace(x : Operation) : Boolean
  x.slotValue(owningClass).memberOperation->includes(self)
```

17.2 SEMANTIC DOMAIN

17.2.1 Derivation

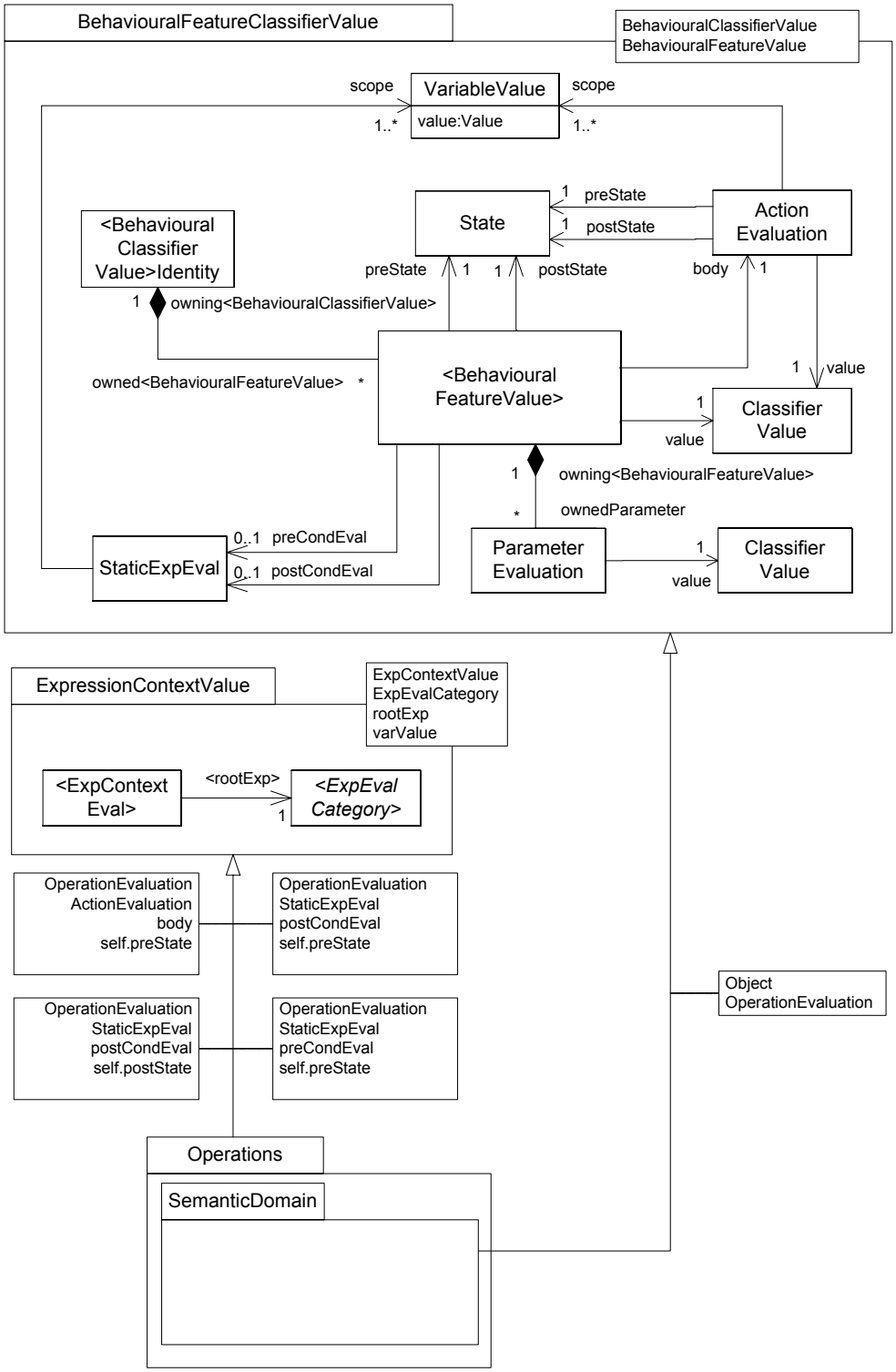


Figure 17-4 *Derivation of Operations semantic domain package*

17.2.2 Model

The semantic domain package for operation is shown in figure 17-5 on page 217 derived as illustrated in figure 17-4 on page 216. An operation instance has a value and is contained by the identity of an object, an operation instance may also has a pre and post condition evaluation and must have a pre and post state. The pre condition evaluation is bound to the environment of the pre state, and the post condition evaluation is bound to the environment of the post state. An operation instance may also have a set parameter evaluations. The body of an operation instance is described by action evaluation which is the root of an action evaluation tree.

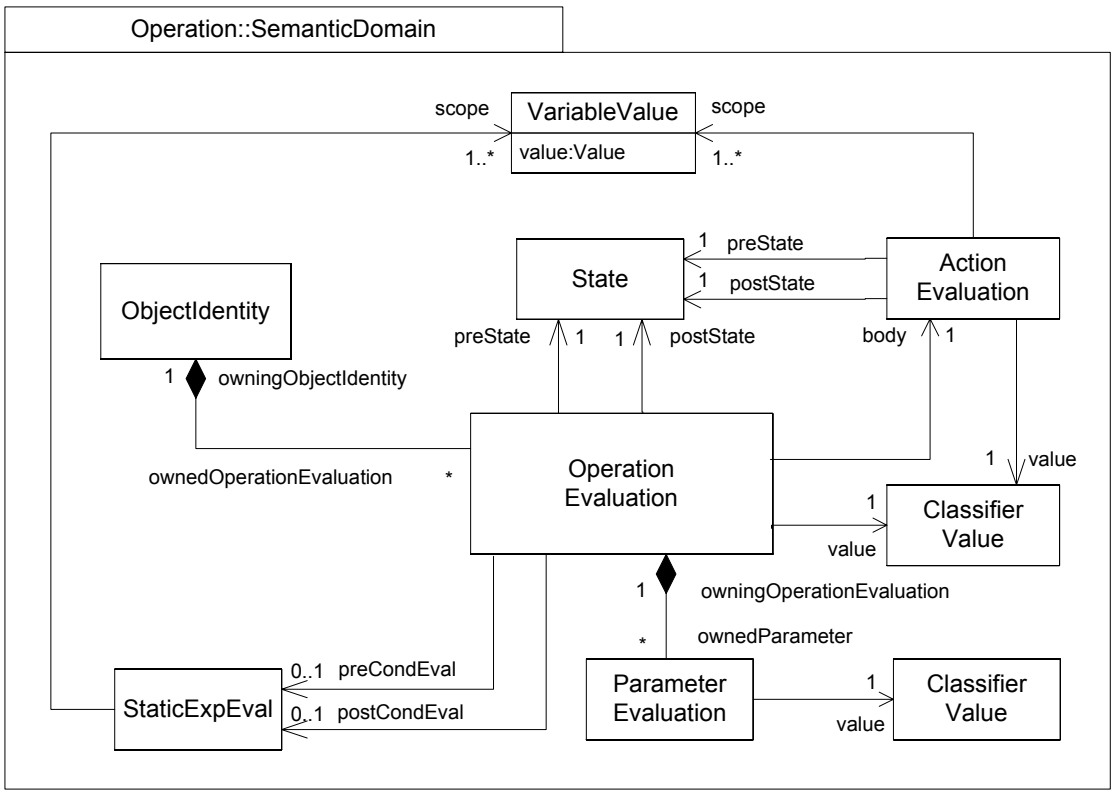


Figure 17-5 *Semantic Domain for Operations package*

ObjectIdentity

Associations

ownedOperationEvaluation The operation evaluations owned by the object identity.

OperationEvaluation

Attributes

preCondEval The evaluation of the operation evaluation's pre condition.

postCondEval The evaluation of the operation evaluation's post condition.

preState The state before the operation evaluation takes place.

postState The state after the operation evaluation takes place.

value The value of the operation evaluation.

body The operation evaluation's body evaluation.

Associations

ownedParameterEvaluation The parameter evaluations of an operation evaluation.

ParameterEvaluation

Attributes

value The value of the parameter evaluation.

Associations

owningOperationEvaluation The operation evaluation owning the parameter.

StaticExpEval

Attributes

value The value of the expression evaluation.

ActionEvaluation

Attributes

preState The state before the action evaluation takes place.

postState The state after the action evaluation takes place.

value The value of the action evaluation.

17.2.3 Well-formedness rules

OperationEvaluation

[1] The post state of an operation evaluation cannot take place before the pre state.

```
context OperationEvaluation inv:
  self.preState.isLater(self.postState)
```

[2] If the pre expression evaluation of an operation evaluation is true, then the post expression evaluation must also be true.

```
context OperationEvaluation inv:
  self.preState.isLater(postState)
```

[3] The pre and post state of an operation evaluation's action evaluation should be the same as self.

```
context OperationEvaluation inv:
  self.body.preState.isSameTime(self.preState) and
    self.body.preState.isSameTime(self.postState)
```

[4] The operation evaluation's action evaluation should have the operation evaluation's pre state in scope.

```
context OperationEvaluation inv:
  self.body.scope->exists(v | v.value=self.preState)
```

[5] The operation evaluation's pre condition should have the operation evaluation's pre state in scope.

```
context OperationEvaluation inv:
  self.preCondEval.scope->exists(v | v.value=self.preState)
```

[6] The operation evaluation's post condition should have the operation evaluation's post state in scope.

```
context OperationEvaluation inv:
  self.postCondEval.scope->exists(v | v.value=self.postState)
```

[7] The operation evaluation's post condition should have the operation's evaluation's pre state in scope.

```
context OperationEvaluation inv:
  self.postCondEval.scope->exists(v | v.value=self.preState)
```

17.2.4 Operations

There are no operations.

17.3 SEMANTIC MAPPING

17.4 DERIVATION

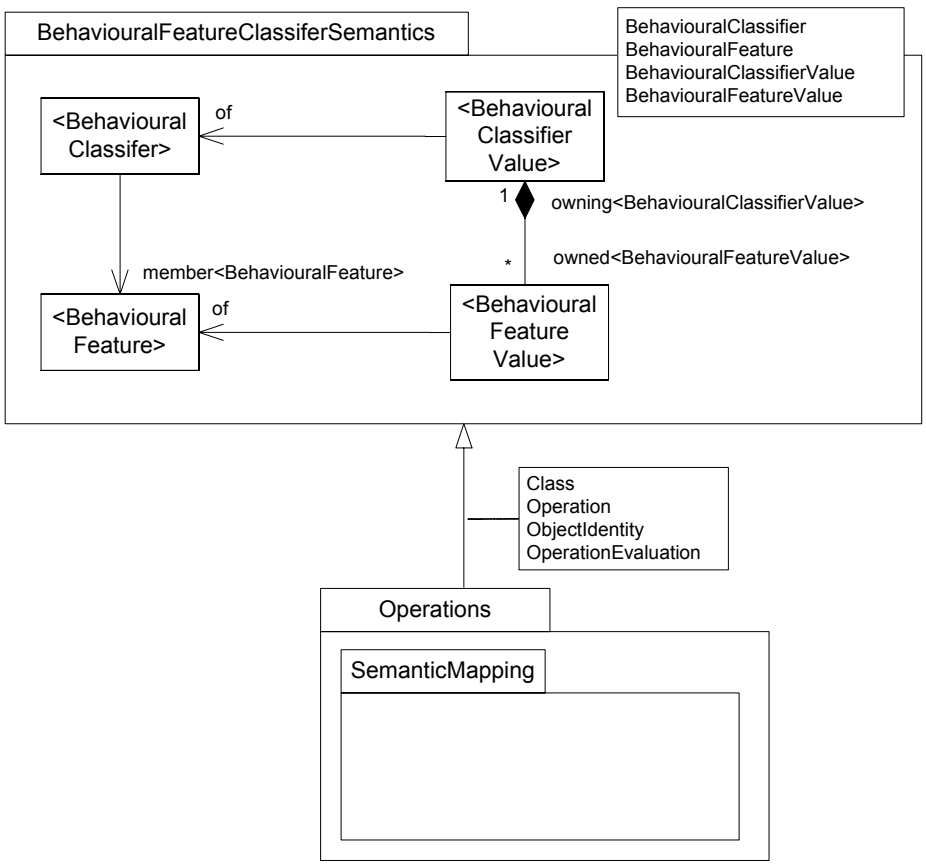


Figure 17-6 *Derivation of semantic mapping for Operations package*

17.4.1 Model

The semantic mapping for the operations package is shown in figure 17-7 on page 220 derived as illustrated in figure 17-6 on page 219. An object identity has an operation evaluation for each of its class's member operations.

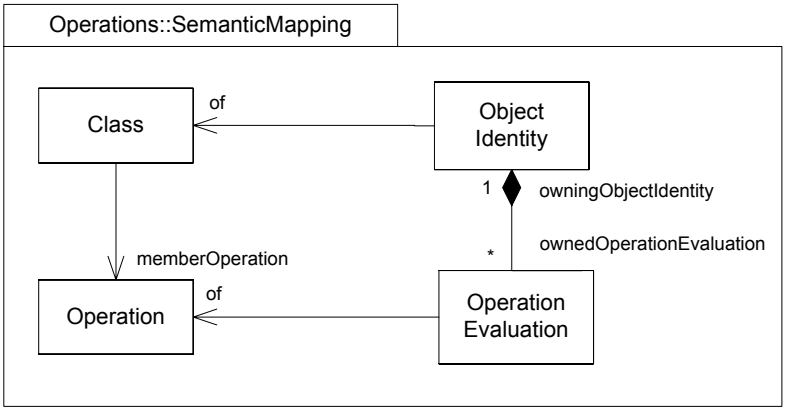


Figure 17-7 *Semantic mapping for Operations package*

Class

Associations

memberOperation The operations that are members of the namespace of the class.

ObjectIdentity

Associations

of The class the object identity is an instance of.

ownedOperationEvaluation The operation evaluations owned by the object identity.

OperationEvaluation

Associations

of The operation the operation evaluation is an instance of.

owningObjectIdentity The object identity owning the operation evaluation.

17.4.2 Well-formedness rules

ObjectIdentity

[1] The object identity's operation evaluations must commute with its class's operation.

```
context ObjectIdentity inv:
  self.ownedOperationEvaluation->forAll(i |
    self.of.memberOperation->exists(o | i.of = o))
```

17.4.3 Operations

There are no operations.

17.5 EXAMPLE SNAPSHOTS

Figure 17-9 on page 221 shows a snapshot realisation of the operation abstract syntax definition shown in figure 17-8 on page 221.

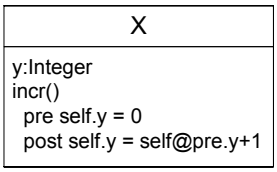


Figure 17-8 *Operation example*

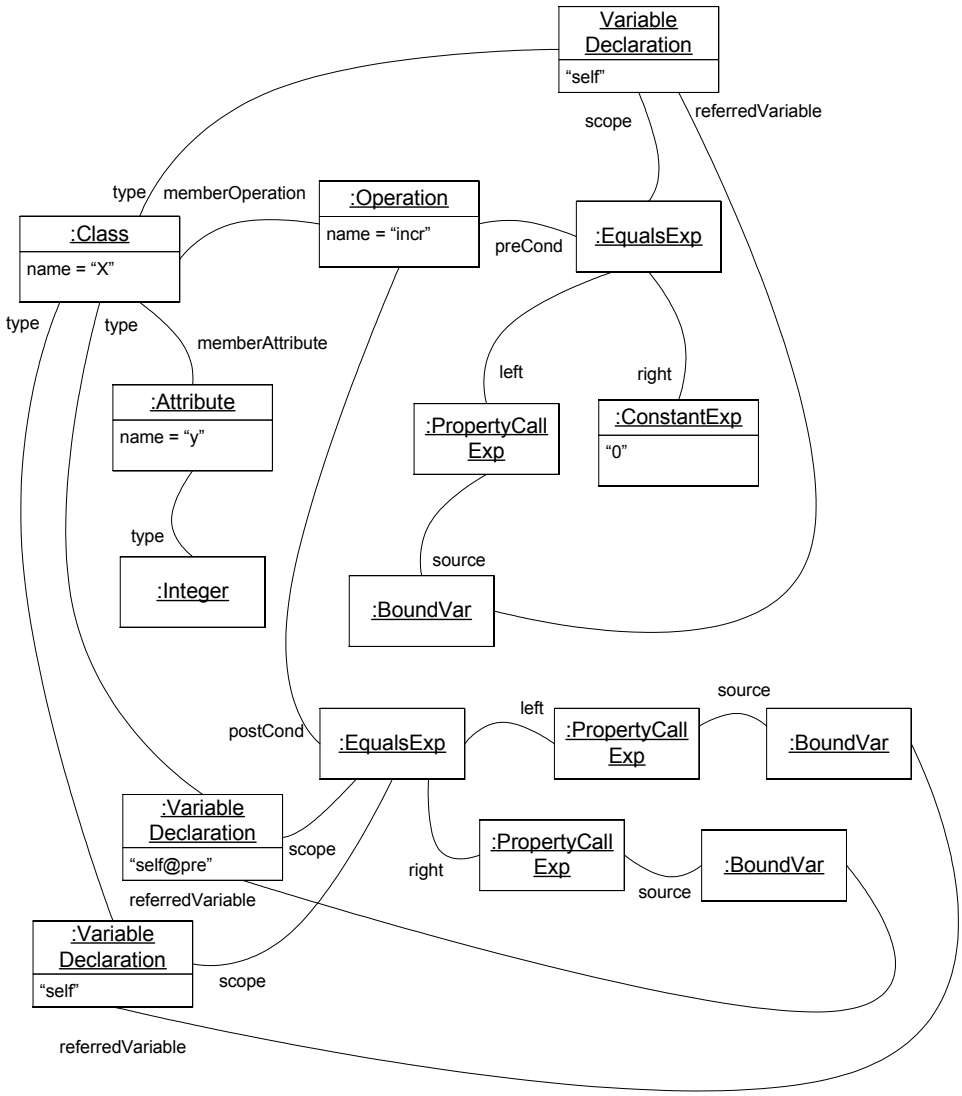


Figure 17-9 *Partial example snapshot of figure 17-3 on page 212*

Although this snapshot is incomplete in as much as we do not include details about the body of the operation (the action tree), it does illustrate how pre and post condition expressions have their scope bound to the class responsible for the operation. For pre expressions, this is simply a binding of the class to self. For post expressions, there is also a binding of the class to self, but in addition there is a binding of the class to self@pre. This enables an instance of a post conditions to reference values within its respective pre condition instance.
Figure 17-10 on page 222 shows a snapshot realisation of the operation semantic domain definition for the syntax specification of figure 17-9 on page 221. Again this is missing details of the operations body, however it is illustrated how the post condition is able to access variable values bound to the pre state (through the semantic realisation of the syntactic self@pre variable declaration).

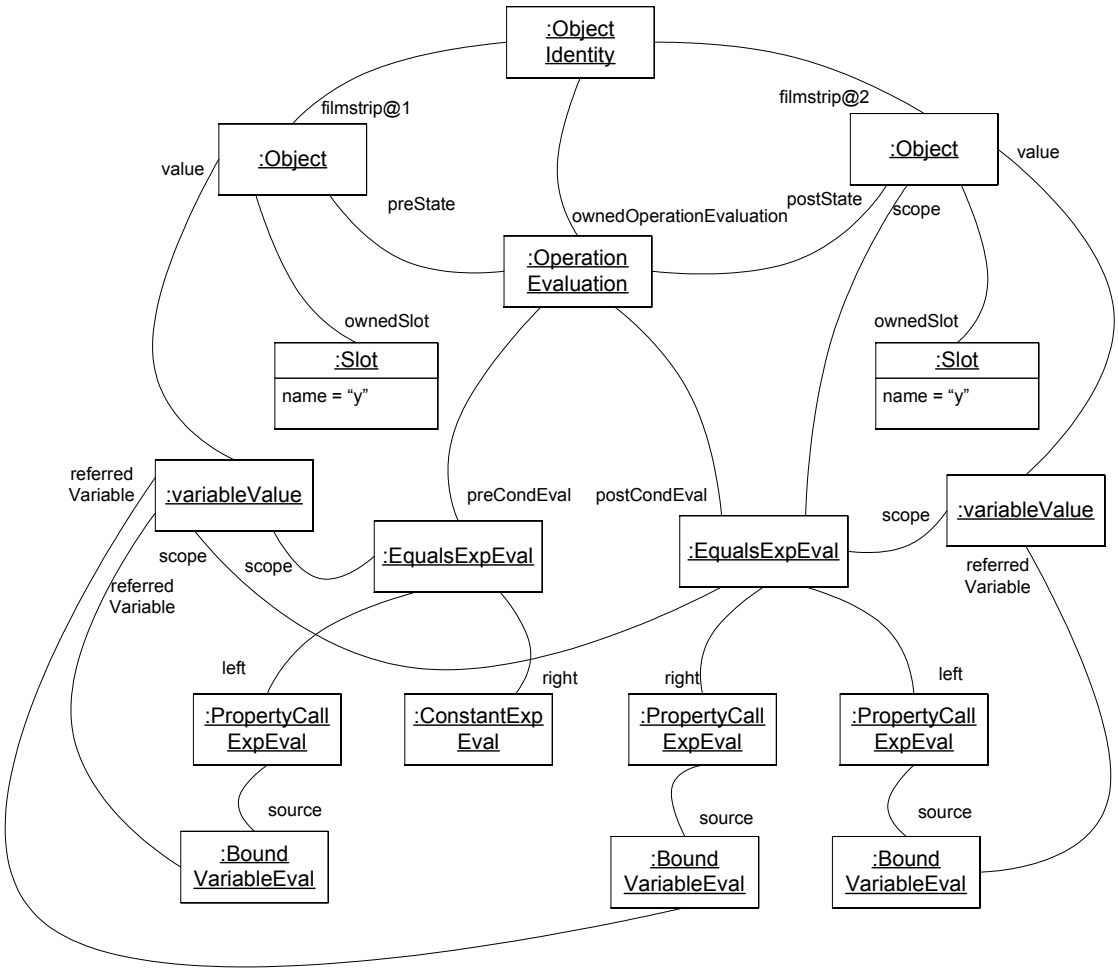


Figure 17-10 *Partial example snapshot of figure 17-5 on page 217*

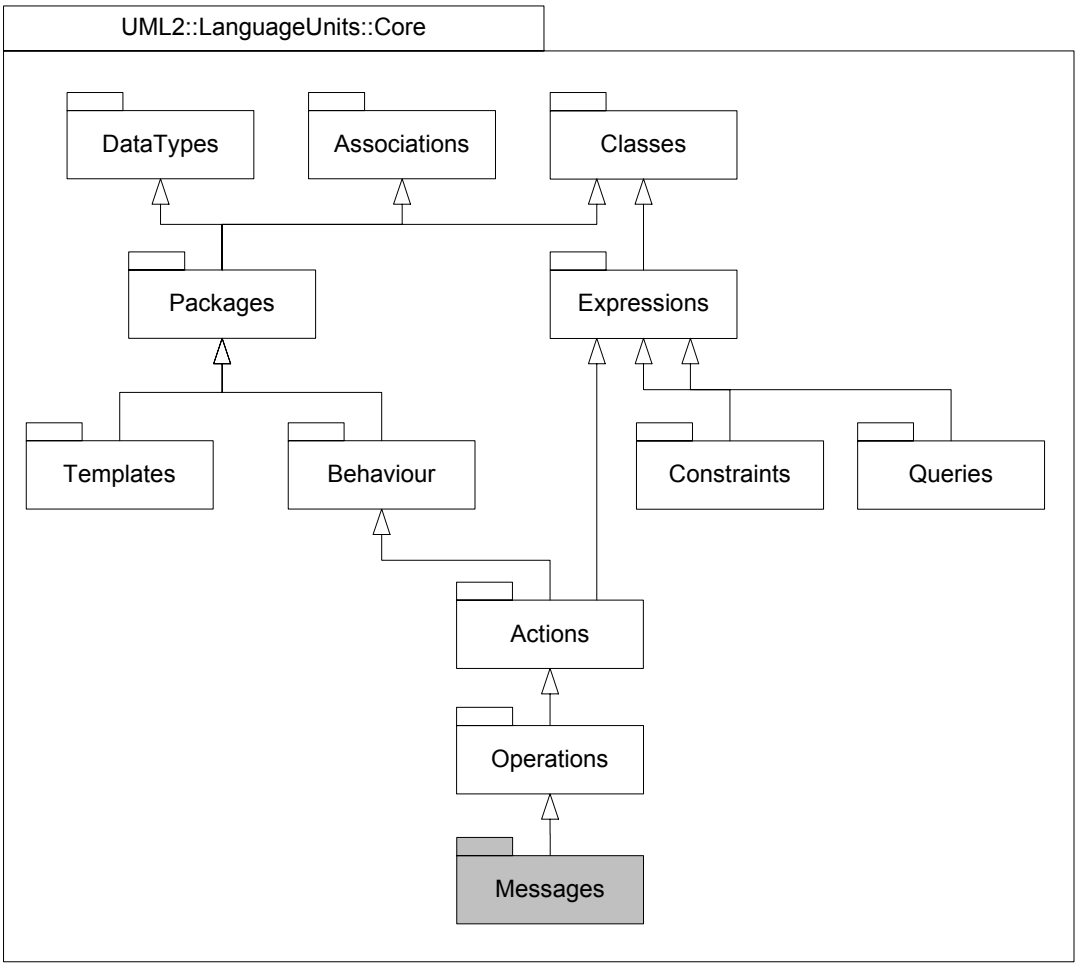
17.6 CHANGES FROM UML 1.4

The semantics for operations have been defined. Operations may be optionally associated with an action, thus supporting a the refinement of operations as action expressions.

Chapter 18
Messaging

This chapter defines an abstract syntax and semantics for messaging. It describes how operations can be invoked by the sending of a message from an object.

18.1 Position in Architecture



The approach we have adopted closely follows that described in (Kleppe01) where objects are augmented with input and output signal queues. When a send message action occurs a new signal is added to the output queue of the object owning the send message action. We say nothing about how the signal is then transferred to the input queue of the target object since this may be realised in a number of ways depending on the target implementation. It is simply stated that if an operation executes then a signal corresponding to invoking the operation must have

been generated sometime earlier in time and that the signal exists in the input queue of the object containing the operation execution and not in its output queue.

18.1.1 Example

Figure 18-1 on page 224 shows an example of a message call (signified by the "^" symbol).

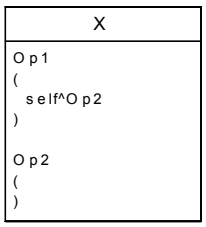


Figure 18-1 *Message call example*

18.2 ABSTRACT SYNTAX

18.2.1 Derivation

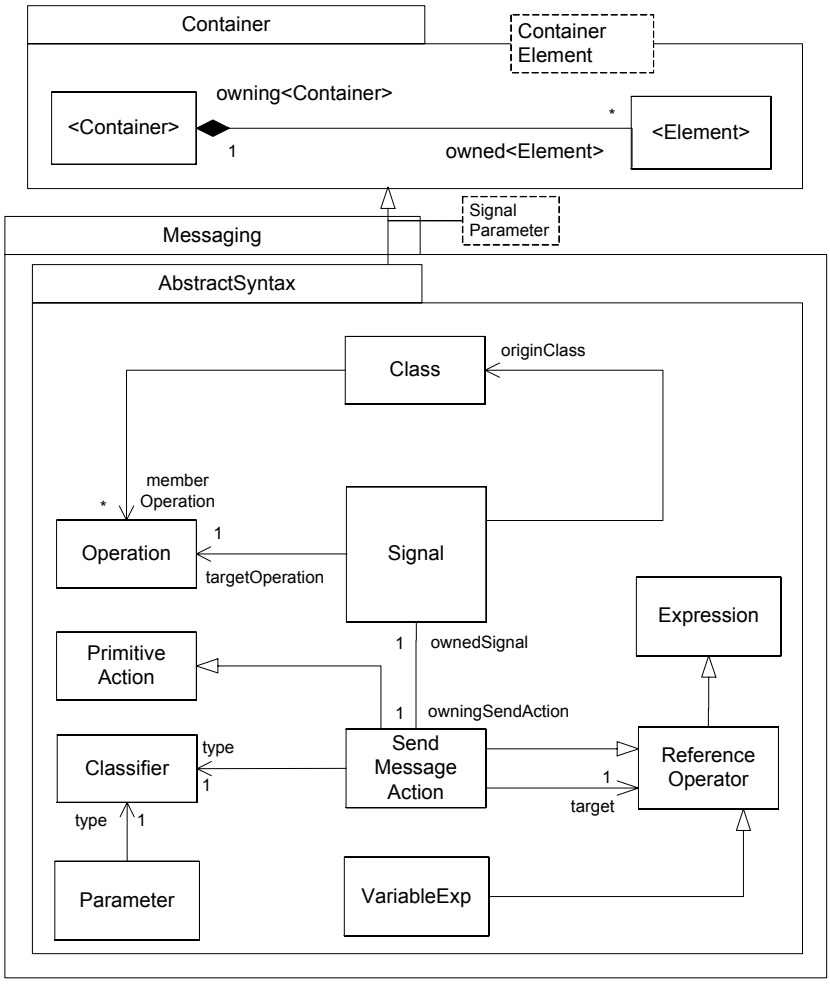


Figure 18-2 *Derivation of Messaging Abstract Syntax package*

18.2.2 Model

Figure 18-3 on page 225 shows the definition of the Messaging package abstract syntax. The derivation of this is illustrated in figure 18-2 on page 224.

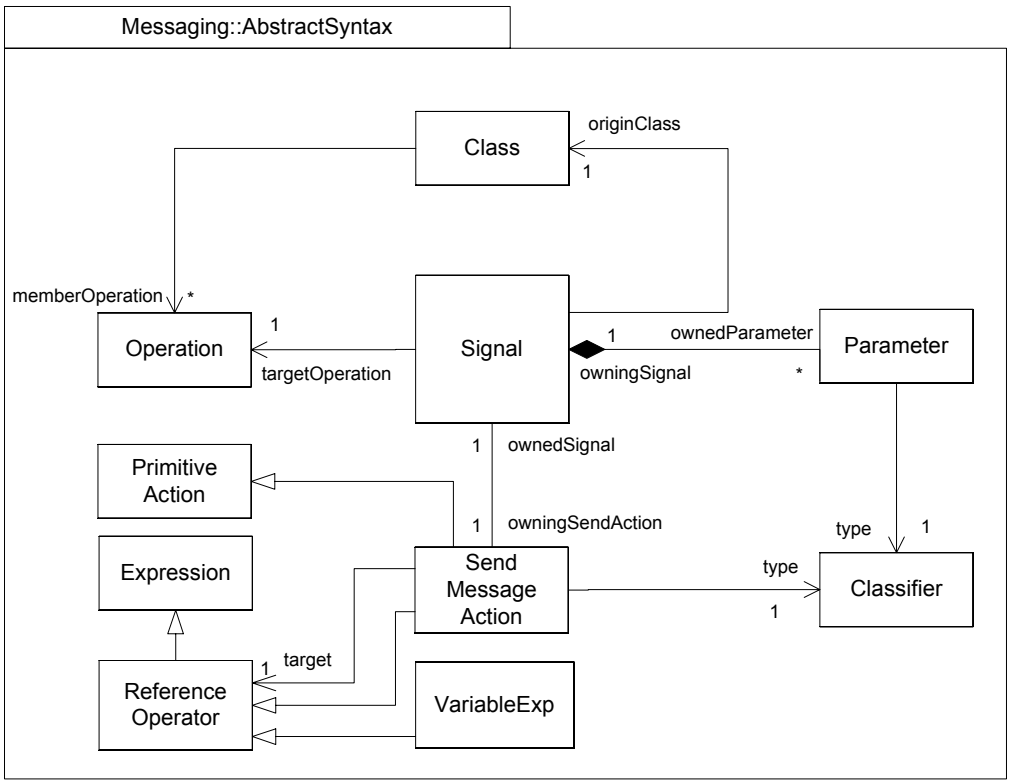


Figure 18-3 *Abstract Syntax for the Messaging package*

Signal

Associations

originClass The class from where the signal originated.

ownedParameter The parameters associated with a signal.

owningSendAction The send message action that initiated the signal.

targetOperation The operation that should be invoked as a result of the signal.

SendMessageAction

Associations

type The type of the send message action.

ownedSignal The signal owned by the send message action.

target The reference operator that links to the target class whose operation needs to be called.

Parameter

Associations

type The type of the parameter.

owningSignal The signal owning the parameter.

ReferenceOperator

ReferenceOperator is an abstract class used purely for the purpose of polymorphism. The target of a SendMessageAction can be a bound variable, a PropertyCallExp or an other SendMessageAction. Using this we can consider the target of a SendMessageAction more generally as a referenceOperator. For example self.a.operation1() is a SendMessageAction with operation1() as the target operation and self.a (a PropertyCallExp expression) as the target.

18.2.3 Well-formedness Rules

SendMessageAction

[1] The type of the SendMessageAction is the return type of the target operation

```
context SendMessageAction inv:
    self.type = self.ownedSignal.targetOperation.type
```

[2] The target operation to be called must be in scope of the target class.

```
context SendMessageAction inv:
    self.target.type.memberOperation ->includes
      (self.ownedSignal.targetOperation)
```

18.2.4 Operations

There are no operations.

18.3 SEMANTIC DOMAIN

18.3.1 Derivation

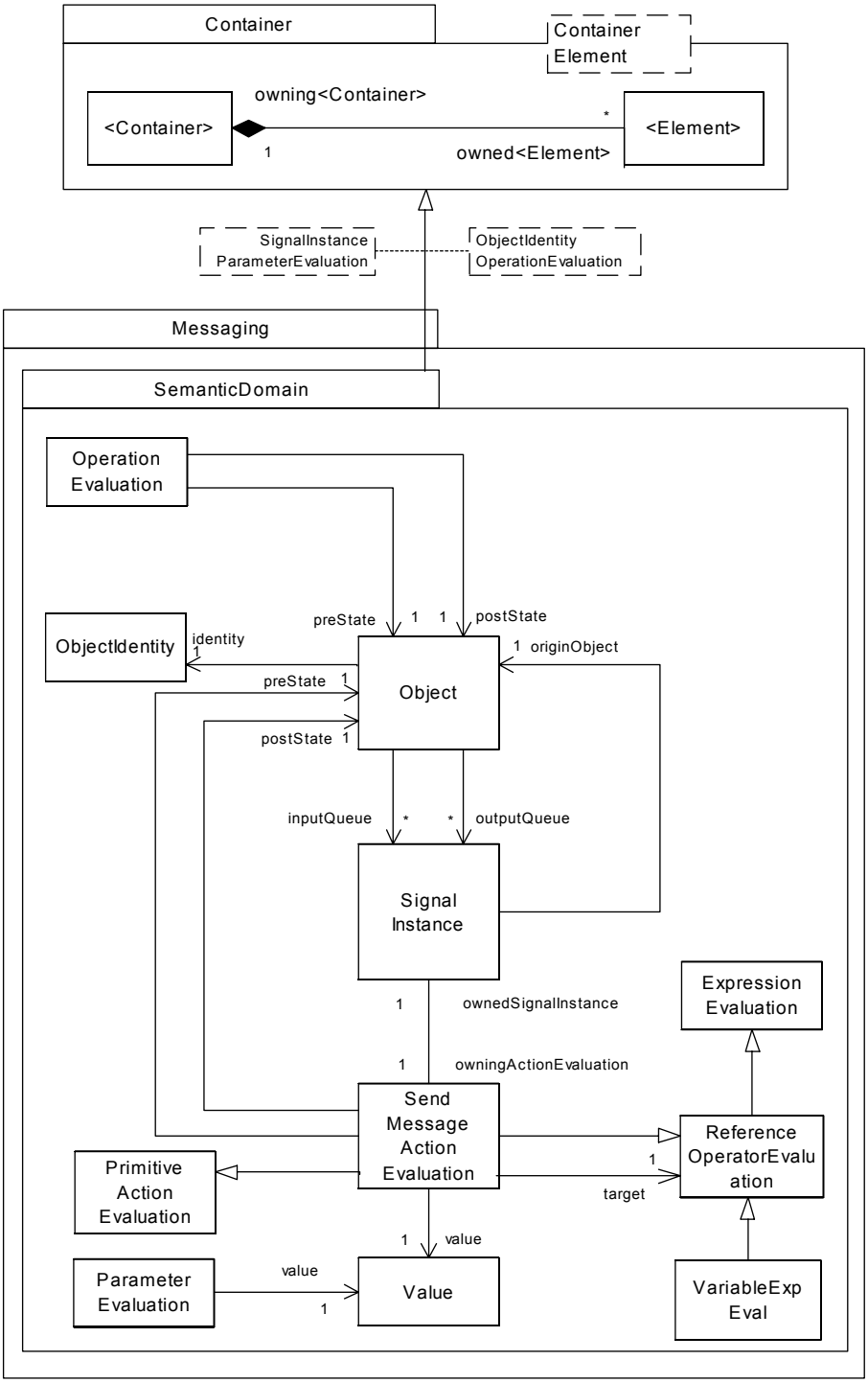


Figure 18-4 *Derivation of Messaging Semantic Domain package*

18.3.2 Model

Figure 18-5 on page 228 shows the definition of Messaging semantic domain package. The derivation of this is illustrated in figure 18-4 on page 227.

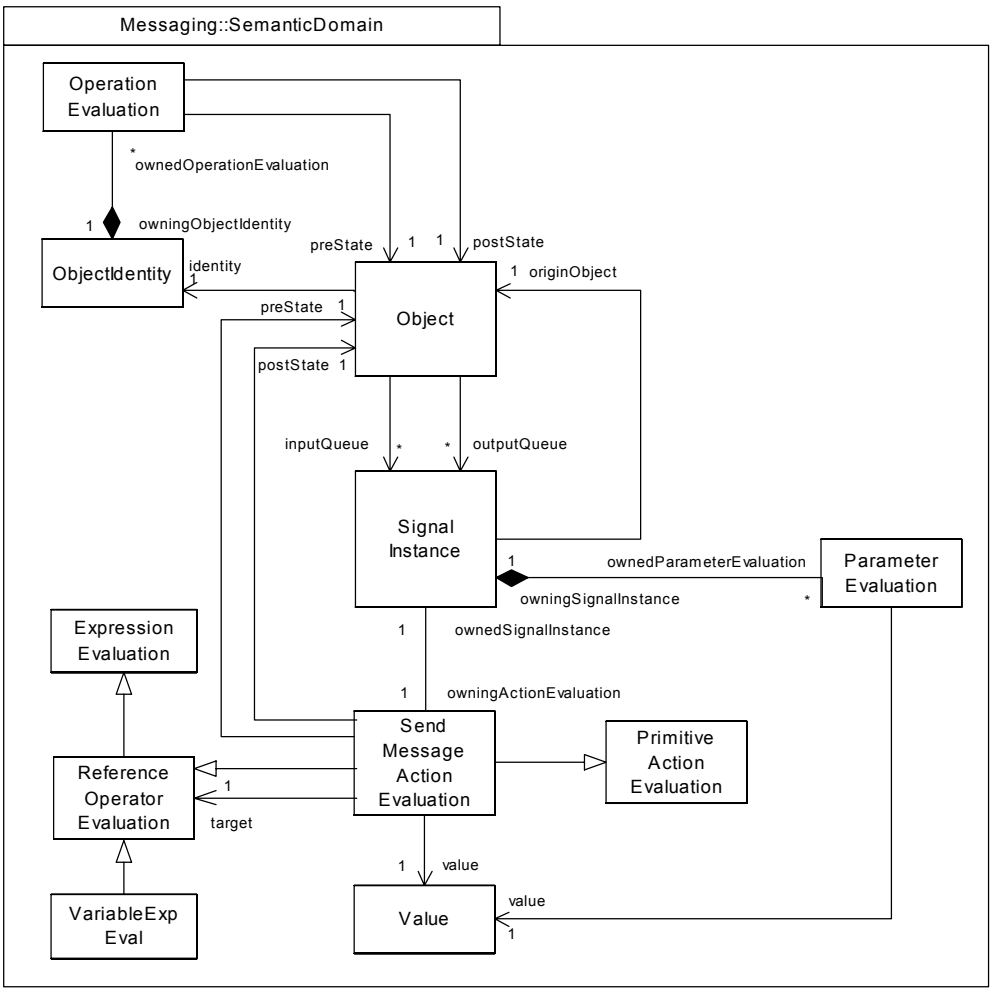


Figure 18-5 *Semantic Domain for Messaging package*

Object

Associations

identity The identity of the object.

inputQueue The signal instances to be processed by the object.

outputQueue The signal instances originating from the object.

ObjectIdentity

Associations

ownedOperationEvaluation The operation evaluations owned by the object identity.

OperationEvaluation

Associations

owningObjectIdentity The object identity owning the operation evaluation.

preState The state before the operation executes.

postState The state after the operation executes.

SignalInstance

Associations

originObject The object from where the signal instance originated.

ownedParameterEvaluation The parameter evaluations owned by the signal instance.

owningActionEvaluation The send message action evaluation owning the signal instance.

SendMessageActionEvaluation

Associations

preState The state before the send message action evaluation takes place.

postState The state after the send message action evaluation has taken place.

value The value of the send message action evaluation.

ownedSignalInstance The signal instance owned my the send message action evaluation.

target The reference operator evaluation that links the target object whose operation needs to be called.

Parameter

Associations

value The value of the parameter.

18.3.3 Well-formedness Rules

SendMessageActionEvaluation

[1] My pre and post states must correspond to the owning object of the send message action evaluation.

```
To be formalised.
```

[2] The pre state and post state must refer to an object with the same identity and correspond to the identity of my signal instances origin object.

```
context SendMessageActionEvaluation inv:
    self.preState.identity = self.postState.identity
      and self.preState.identity =
        self.ownedSignalInstance.originObject.identity
```

[3] My signal instance must not be included in my pre state object's output queue, but should be included in my post state's output queue.

```
context SendMessageActionEvaluation inv:
   not(self.preState->includes(ownedSignalInstance)) and
     self.postState->includes(ownedSignalInstance)
```

Object

[1] The signal instances in the input queue no longer exist in the output queue of their origin object.

```
context Object inv:
    self.inputQueue->forAll(i |
      not(i.originObject->outputQueue->includes(i))
```

18.3.4 Operations

There are no operations.

18.4 SEMANTIC MAPPING

18.4.1 Derivation

There is no derivation.

18.4.2 Model

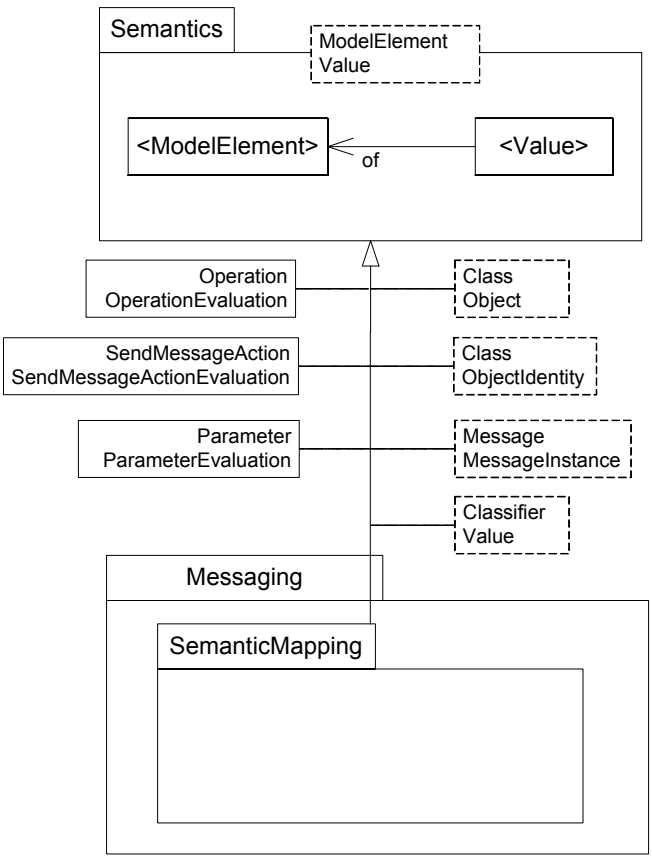


Figure 18-6 *Derivation of Messaging Semantic Mapping package*

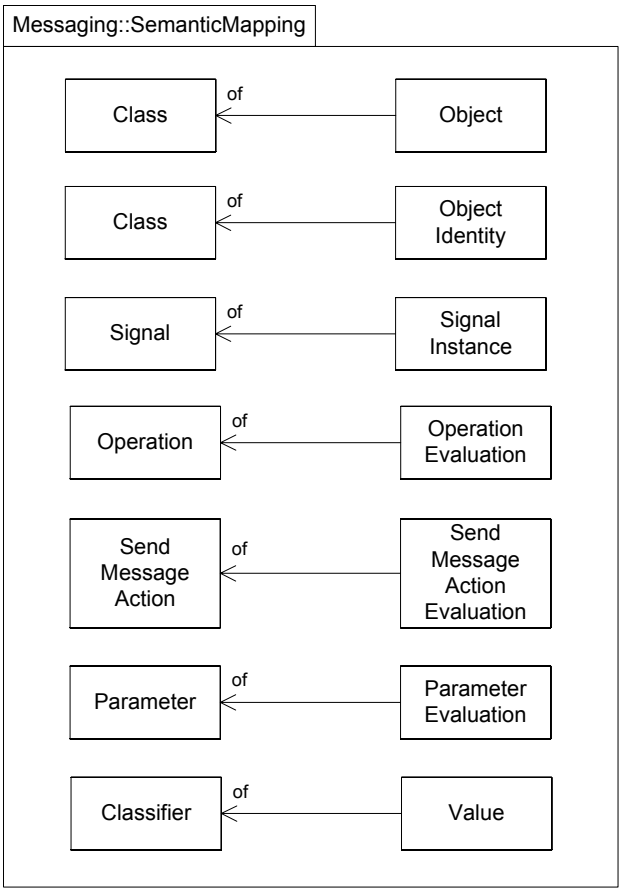


Figure 18-7 *Semantic Mapping for Messaging package*

18.4.3 Well-formedness Rules

OperationEvaluation

[1] There must exist in the pre state object's input queue a signal instance who targets the operation that I am an instance of. The signal should have been created earlier in time. This signal should not exist in the post state of my object's input queue.

```
context OperationEvaluation inv:
  self.preState.inputQueue->includes(i | i.of.targetOperation = self.of
    and self.preState.isLater(i.owningActionEvaluation.postState)
      and not(self.postState.inputQueue->includes(i)))
```

[3] My object identity's class should contain an operation that commutes with me.

```
context OperationInstance inv:
  self.owningObjectIdentity.of.
     memberOperation->includes(i | self.of = i)
```

18.4.4 Operations

There are no operations.

18.5 EXAMPLE SNAPSHOTS

Figure 18-9 on page 232 shows a snapshot realisation of the example shown in figure 18-8 on page 232.

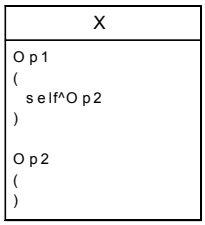


Figure 18-8 *Example message*

This describes how a class has two operation (Op1 and Op2) and how the second operation (Op2) is invoked from the first (Op1).

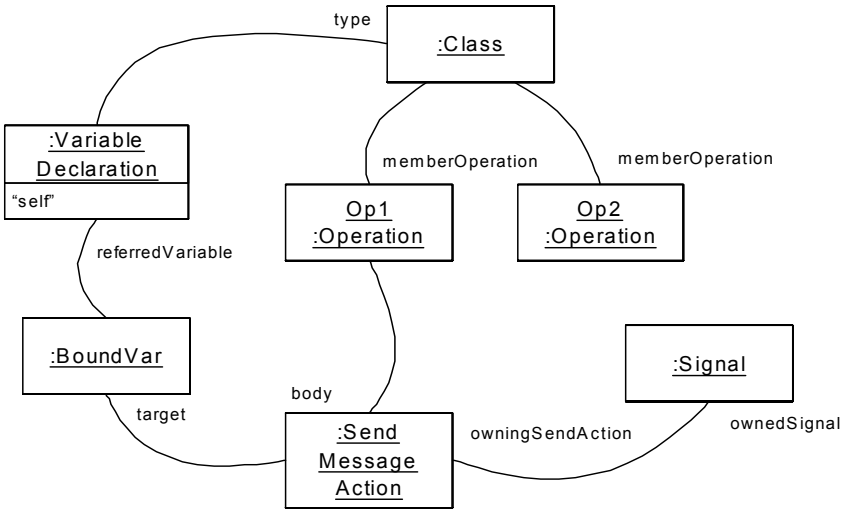


Figure 18-9 *Example snapshot of figure 18-3 on page 225*

Figure 18-10 on page 233 shows a snapshot realisation of the messaging semantic domain definition for the syntax specification of figure 18-9 on page 232. The evolution of the system is described such that the there is an object has a signal in the post state of the send message action (filmstrip@2) that did not exist in the pre state (filmstrip@1). This state transformation was ultimately caused by the first operation (Op1). The second operation (Op2, which does nothing) occurs later in time and describes how the same signal exists in the input queue of its object and no longer exists in the output queue of the origin object (the same object) and that signal does not exist after the operation has executed.

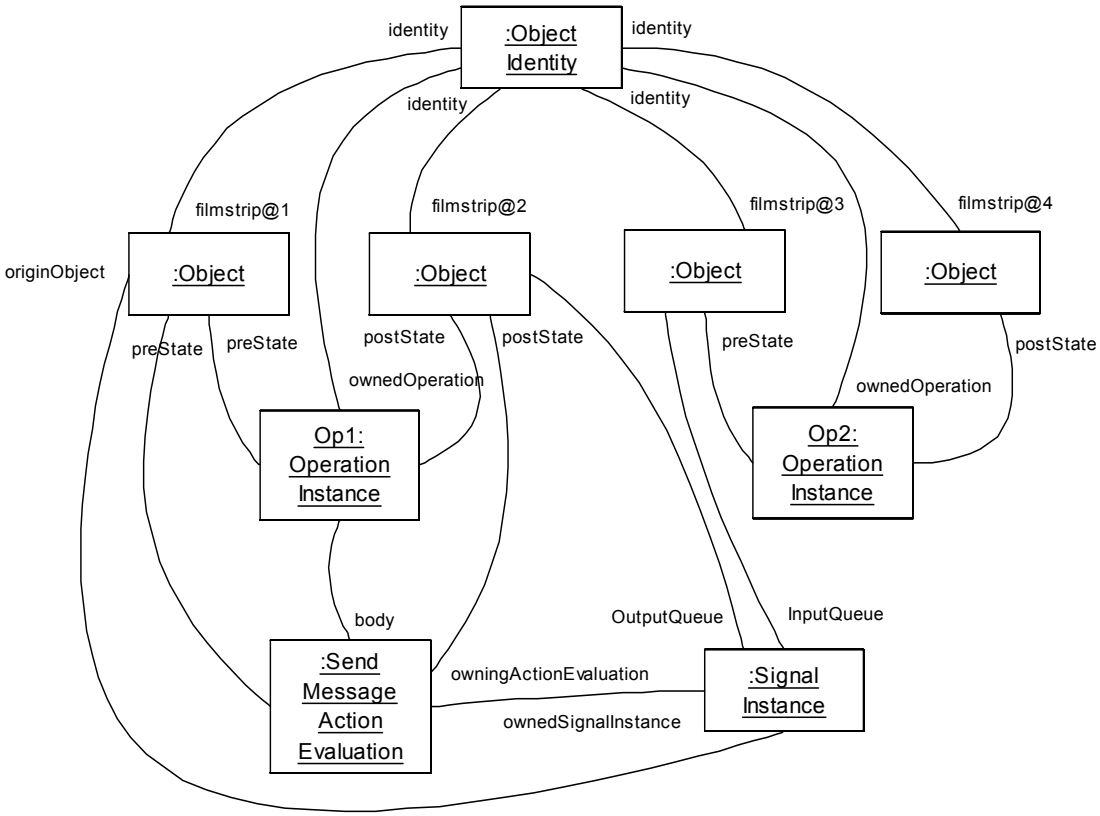


Figure 18-10 *Example snapshot of figure 18-5 on page 228*

18.6 CHANGES TO UML 1.4

A semantics has been defined for message passing.

Chapter 19
Foundation Templates

19.1 INTRODUCTION

The purpose of this chapter is to describe a set of general purpose templates for language design. Each of the templates described in this chapter represent a self-contained unit of concepts and properties that capture a specific aspect of language design. These templates are used to construct the UML specific templates that can be found in the next chapter.

The templates in this chapter are categorised and ordered as follows:

Structural Templates: Container, TypedElement, Parameterized, Multiplicity.

Naming Templates: Named, Namespace.

Relationship: Relationship, Generalizable, Extendable, Import.

Semantics: Semantics, ParameterizedValue, ParameterizedSemantics.

These templates and categories are not fixed. In the process of building the submission, we have noticed many other useful language design templates. Our intention is to expand this chapter with new templates as we identify them and our experience of language definition grows.

19.2 CONTAINER

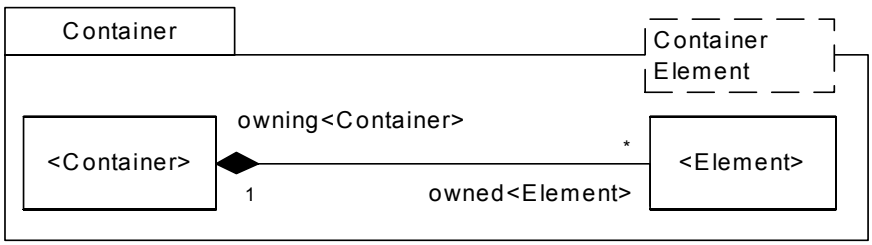
19.2.1 Summary

A containment relationship, in which one element, the container, conceptually contains another element (the contained element). Containers are one of the most fundamental patterns found in a modelling language. Many language elements “contain” other language elements.

19.2.2 Derivation

Not derived from any template.

19.2.3 Definition



<Container>

Associations

owned<Element> The set of owned/contained elements.

<Element>

Associations

owning<Container> The container which owns/contains the element.

19.2.4 Well-formedness Rules

19.2.5 Operations

19.3 TYPEDELEMENT

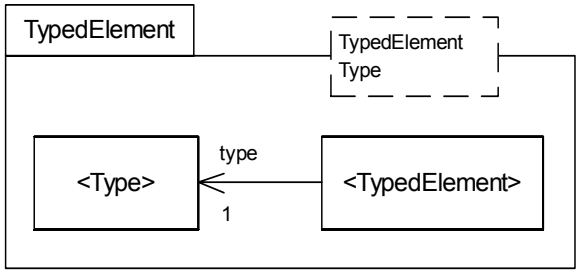
19.3.1 Summary

This template defines the structure of elements that have a type.

19.3.2 Derivation

Not derived from any template.

19.3.3 Definition



<TypedElement>

Associations

type The type of the typed element.

19.3.4 Well-formedness Rules

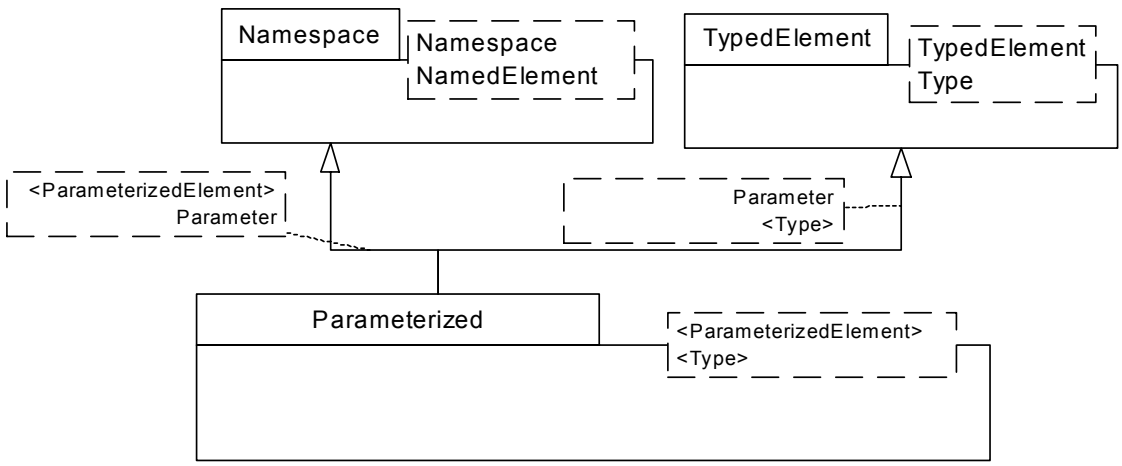
19.3.5 Operations

19.4 PARAMETERIZED

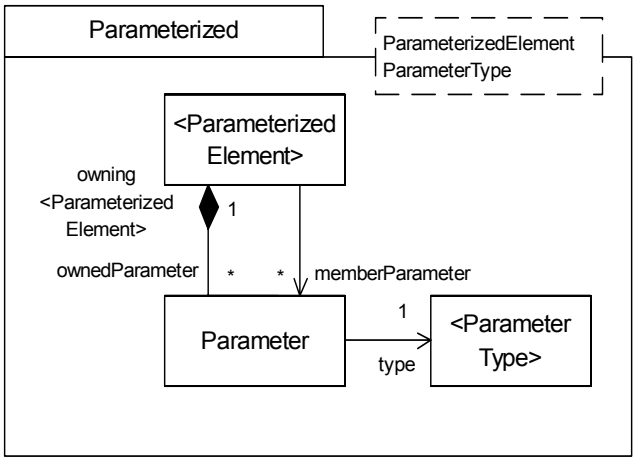
19.4.1 Summary

An element which has typed parameters.

19.4.2 Derivation



19.4.3 Definition



<ParameterizedElement>

Association

memberParameter The members of the parameterized element's namespace.

ownedParameter The owned parameters of the parameterized element.

Parameter

Associations

type The type of the parameter.

19.4.4 Well-formedness Rules

<ParameterizedElement>

[1] The members of a parameterized element include its owned parameters.

```
context <ParameterizedElement> inv:
  self.memberParameter->includesAll(self.ownedParameter)
```

[2] A parameterised element cannot have two parameters with the same name.

```
context <ParameterizedElement> inv:
  self.memberParameter->forAll(e1|
    self.memberParameter->forAll(e2|
      e1 <> e2 implies e1.name <> e2.name))
```

19.4.5 Operations

<ParameterizedElement>

[1] Looks up a parameter in a parameterized element given a name.

```
context <ParameterizedElement>::lookupParameterforName(x : Name): Parameter
  self.memberParameter->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a parameterized element given a parameter.

```
context <ParameterizedElement>::lookupNameForParameter(n : Parameter):Name
  self.memberParameter->select(e | e = x).selectElement().name
```

Parameter

[1] Checks whether the given parameter is in the same namespace as this namespace

```
context Parameter::sameNamespace(x : Parameter):Boolean
  x.owning<ParameterizedElement>.memberParameter -> includes(self)
```

19.5 MULTIPLICITY

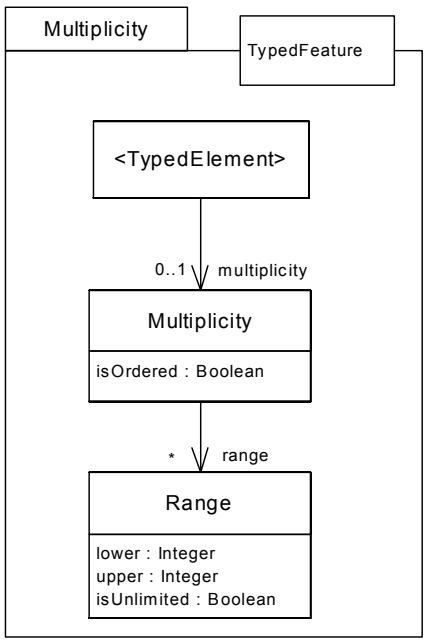
19.5.1 Summary

A multiplicity is a set of integer values including the distinguished value "unLimited". A multiplicity is associated with a range which specifies the range of integer values in the set.

19.5.2 Derivation

Not derived from any template.

19.5.3 Definition



<TypedFeature>

Attributes

multiplicity The multiplicity associated with the typed feature.

Multiplicity

Attributes

isOrdered True if the elements are to be ordered.

range The set of number ranges belonging to the multiplicity.

Range

Attributes

lower The lower value

upper The upper value

isUnlimited True if the range is infinite

19.5.4 Well-formedness Rules

19.5.5 Operations

19.6 NAMED

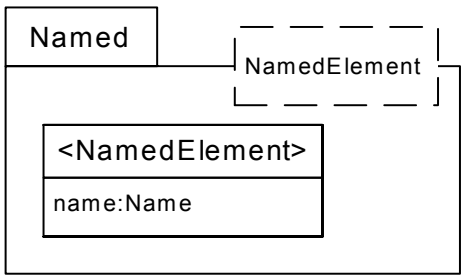
19.6.1 Summary

A named element.

19.6.2 Derivation

Not derived from any template.

19.6.3 Definition



<NamedElement>

attribute

name The name of the named element

19.6.4 Well-formedness Rules

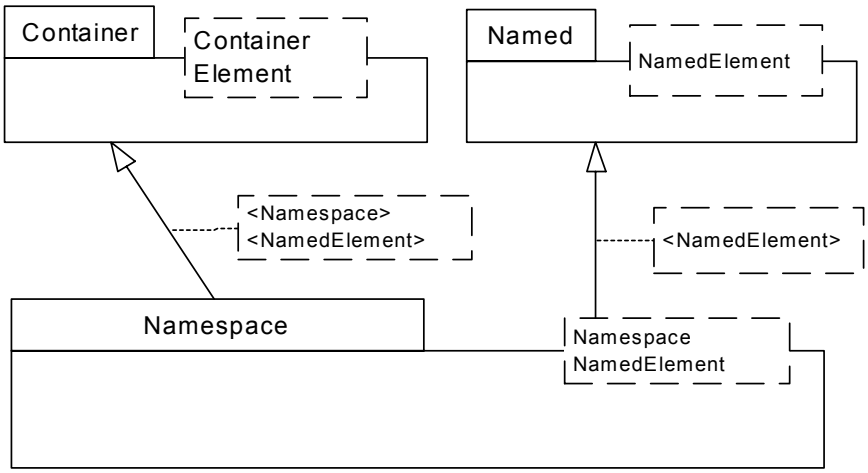
19.6.5 Operations

19.7 NAMESPACE

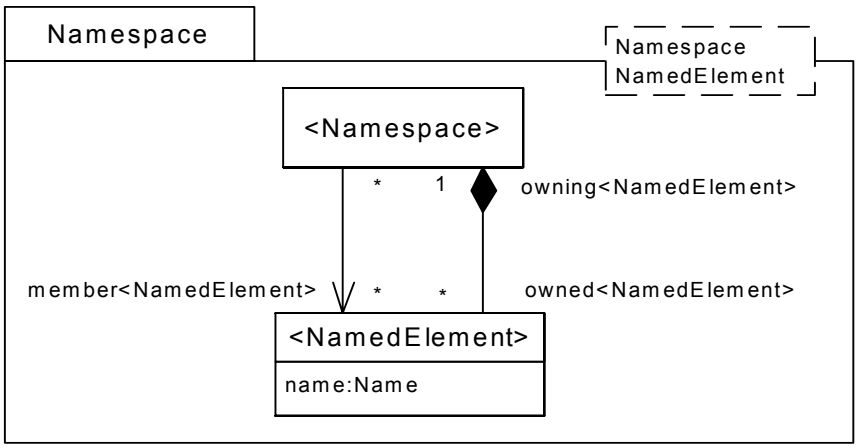
19.7.1 Summary

A namespace for named elements. A named element is a member of a namespace if it is owned by the namespace or has been included as a result of import, extension or inheritance. A namespace provides lookup operations that return a named element for a name and vice versa.

19.7.2 Derivation



19.7.3 Definition



<Namespace>

Attributes

member<NamedElement> The members of the namespace.

Associations

owned<NamedElement> The owned named elements.

<NamedElement>

Attributes

name The name of the named element.

Associations

owningNamespace The namespace owning the named element.

19.7.4 Well-formedness Rules

<Namespace>

[1] The members of a namespace include its owned elements

```
context <Namespace> inv:
  self.member<NamedElement>->includesAll(self.owned<NamedElement>)
```

[2] A namespace cannot have two named elements with the same name.

```
context <Namespace> inv:
  self.member<NamedElement>->forAll(e1|
    self.member<NamedElement>->forAll(e2|
      e1 <> e2 implies e1.name <> e2.name))
```

19.7.5 Operations

<Namespace>

[1] Looks up a named element in a namespace given a name

```
context <Namespace>::lookup<NamedElement>forName(x : Name): <NamedElement>
  self.member<NamedElement>->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a namespace given a named element

```
context <Namespace>::lookupNameFor<NamedElement>(n : <NamedElement>):Name
  self.member<NamedElement>->select(e | e = x).selectElement().name
```

<NamedElement>

[1] Checks whether the given named element is in the same namespace as this namespace

```
context <NamedElement>::sameNamespace(x : <NamedElement>):Boolean
  x.owning<Namespace>.member<NamedElement> -> includes(self)
```

19.8 RELATIONSHIP

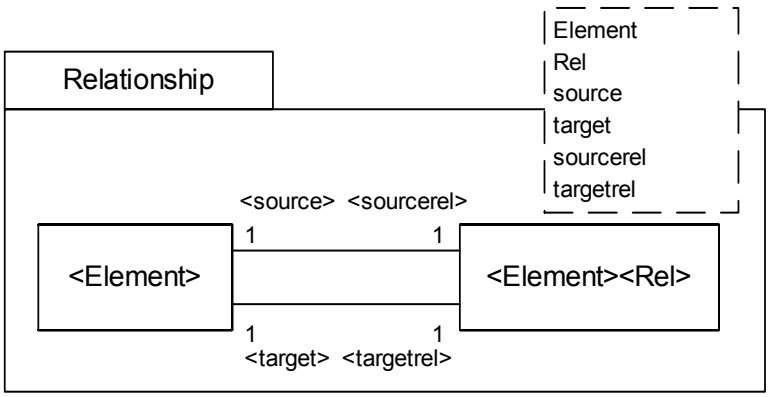
19.8.1 Summary

Defines a relationship between two elements of the same type.

19.8.2 Derivation

Not derived from any template.

19.8.3 Definition



<Element>

assocoation

<*sourcerel*> The source elements.

<*targetrel*> The target elements.

<Element><Rel>

association

<*source*> The source element.

<*target*> The target element.

19.8.4 Well-formedness Rules

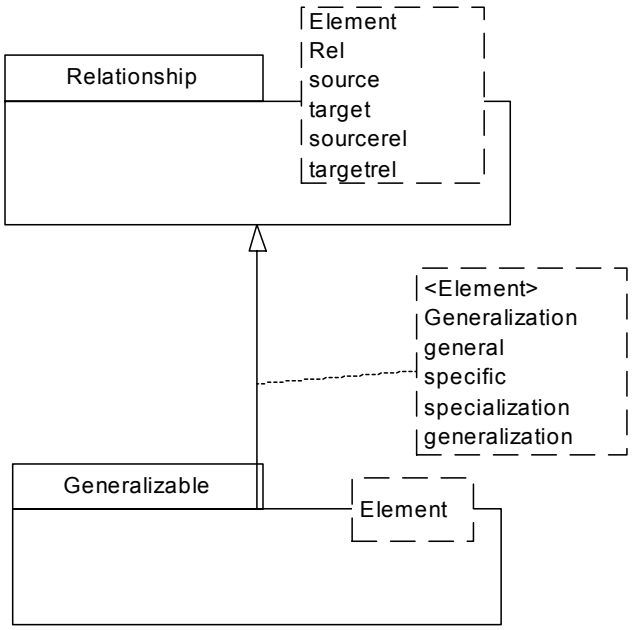
19.8.5 Operations

19.9 GENERALIZABLE

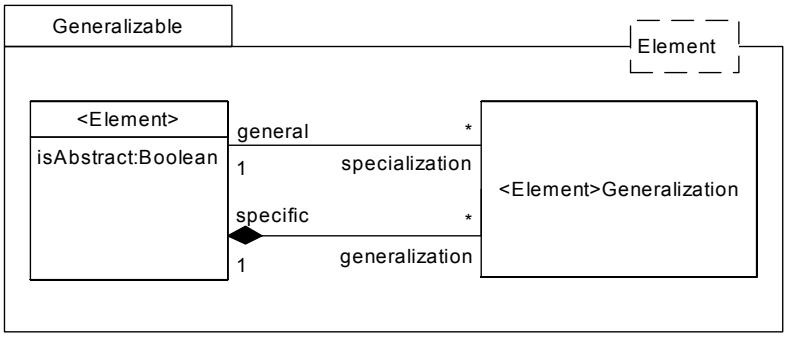
19.9.1 Summary

A generalization relationship between elements.

19.9.2 Derivation



19.9.3 Definition



<Element>

Attributes

isAbstract True if the element is abstract.

Associations

specialization The specializations of element.

generalization The generalizations of element.

<Element>Generalization

Associations

general The general element.

specific The specific element.

19.9.4 Well-formedness Rules

<Element>

[1] Circular inheritance is not permitted

```
context <Element> inv:
  not self.allGeneralElements()->includes(self)
```

19.9.5 Operations

<Element>

[1] Returns the generalizations of the element.

```
context <Element>::generalElements():Set(<Element>)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[2] Transitively returns all generalizations of the element.

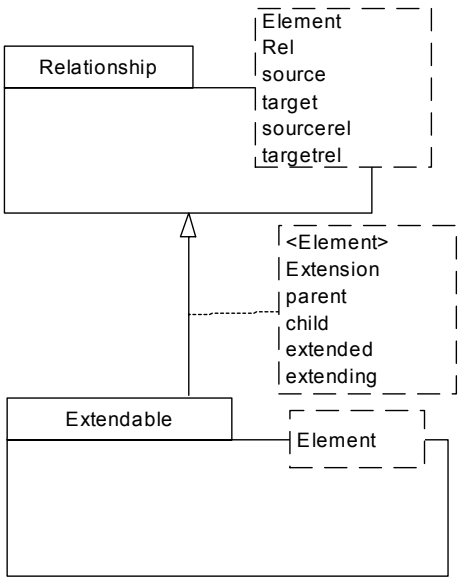
```
context <Element>::allGeneralElements():Set(<Element>)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

19.10 EXTENDABLE

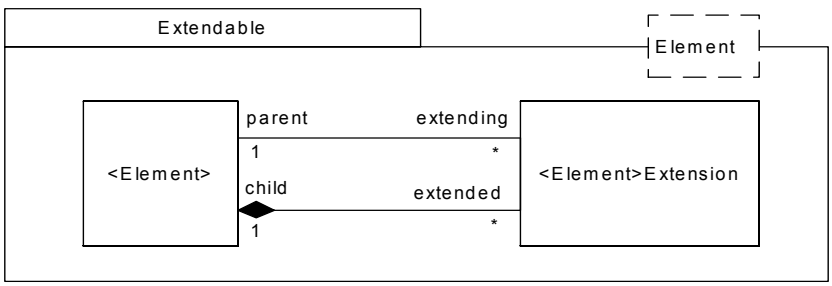
19.10.1 Summary

An extension relationship between elements.

19.10.2 Derivation



19.10.3 Definition



<Element>

Associations

extended The extended elements.

extending The extending elements.

<Element>Extension

Associations

parent The parent Element.

child The child Element

19.10.4 Well-formedness Rules

<Element>

[1] Circular inheritance is not permitted.

```
context <Element> inv:
  not self.allExtendedElements()->includes(self)
```

19.10.5 Operations

<Element>

[1] Returns the elements that have been extended.

```
context <Element>::extendedElements():Set(<Element>)
  self.extended -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[2] Transitively returns all elements that have been extended.

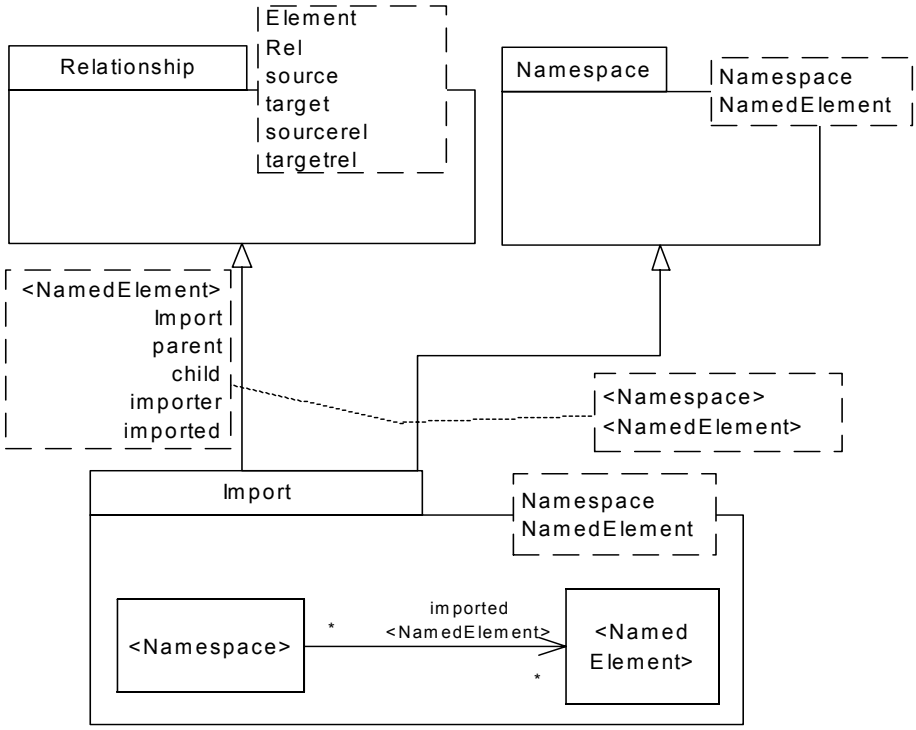
```
context <Element>::allExtendedElements():Set(<Element>)
  self.extendedElements()->iterate(g s = self.extendedElements() |
    s->union(g.allExtendedElements()))
```

19.11 IMPORT

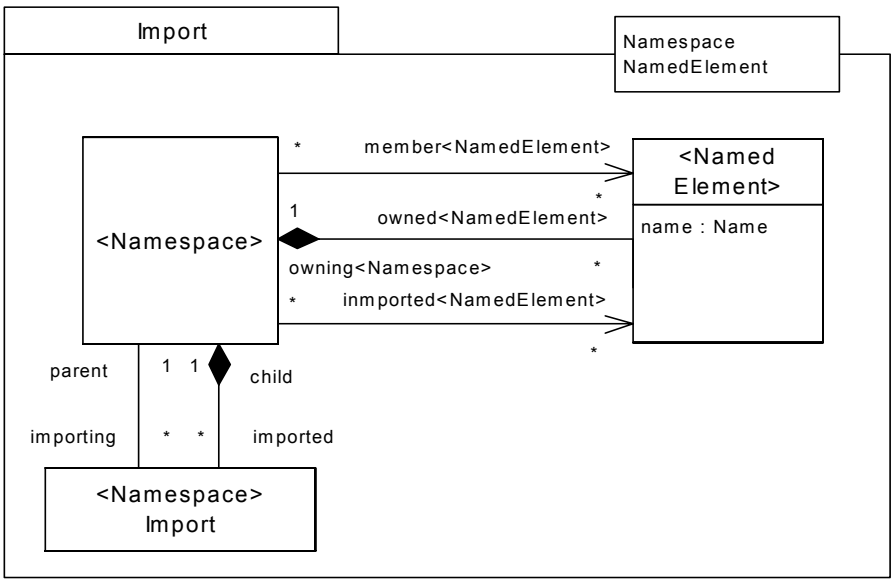
19.11.1 Summary

Defines an import relationship for a pair of namespaces.

19.11.2 Derivation



19.11.3 Definition



<NamedElement>

Associations

imported<*NamedElement*> The imported elements.

member<*NamedElement*> The member elements.

19.11.4 Well-formedness Rules

[1] The members of a namespace include its imported elements

```
context <Namespace> inv:
  self.member<NamedElement>->includesAll(self.imported<NamedElement>)
```

[2] Parent namespace named elements are imported.

```
context <Namespace> inv:
  self.importedNamespaces()->forAll(x |
    self.imported<NamedElement>->includesAll(x.member<NamedElement>))
```

19.11.5 Operations

<Namespace>

[1] Returns the imported namespaces of the namespace.

```
context <Namespace>::imported<Namespace>():Set(<Namespace>)
  self.imported->iterate(p s=Set{} | s->union(Set{p.parent}))
```

[2] Transitively returns all imported namespaces of the namespace.

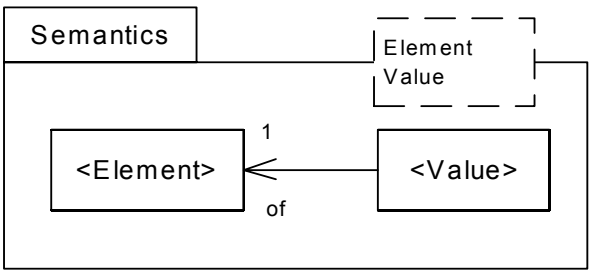
```
context <Namespace>::allImported<Namespace>():Set(<Namespace>)
  self.imported<Namespace>()->iterate(g s=self.imported<Namespace>() |
         s->union(g.allImported<Namespace>()))
```

19.12 SEMANTICS

19.12.1 Summary

An semantic relationship between a value and the element it is a value or instance of.

19.12.2 Model

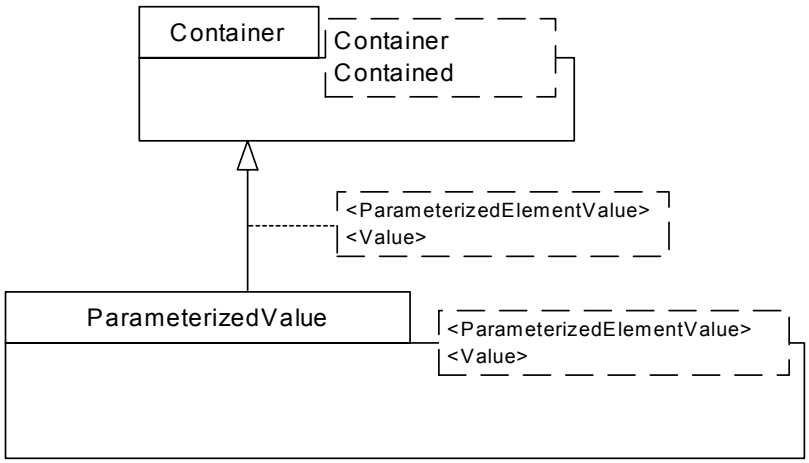


19.13 PARAMETERIZEDVALUE

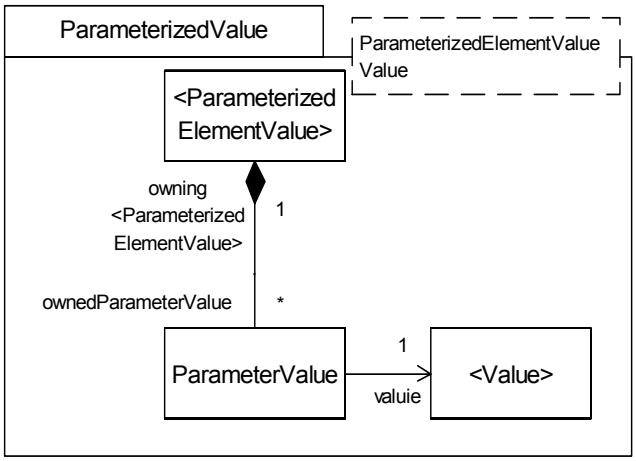
19.13.1 Summary

An instance of a parameter.

19.13.2 Definition



19.13.3 Definition



<ParameterizedElementValue>

Association

ownedParameterValue The owned parameter values of the parameterized element value.

ParameterValue

Associations

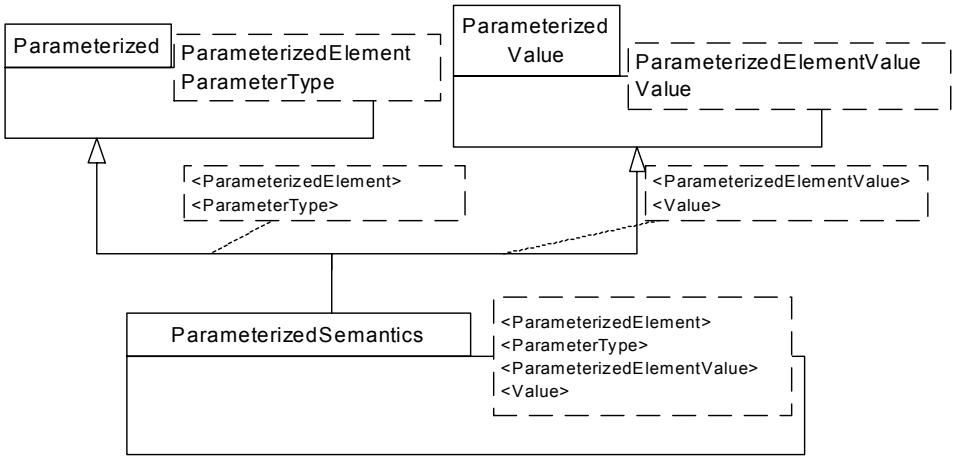
value The value of the parameter value.

19.14 PARAMETERIZEDVALUESEMANTICS

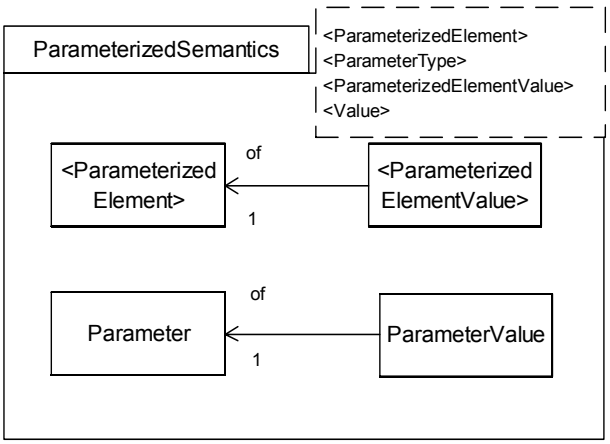
19.14.1 Summary

Defines a semantics for parameterized element. A value of a parameterized element is a parameter value. There must be a parameter value for every parameter of the parameterized element and vice versa.

19.14.2 Derivation



19.14.3 Definition



19.14.4 Well-formedness rules

<ParameterizedElementValue>

[1] A parameterized element value should contain a parameter value for all parameter's in the parameterized element value's parameterized element's namespace.

```
context <ParameterizedElementValue> inv:
  self.of.memberParameter->forAll(c |
    self.ownedParameterValue->exists(d | d.of = c))
```

[2] For each parameter value owned by a parameterized element value there should be a parameter of the parameterized element value's parameter element's namespace that the parameterized element value is a value of.

```
context <ParameterizedElementValue> inv:
  self.ownedParameterValue->forAll(c |
    self.of.memberParameter->exists(d | c.of = d))
```

Chapter 20
UMLTemplates

20.1 INTRODUCTION

This chapter describes the templates used to define UML 2.0. Note, these templates are specifically targeted at the UML language.

The templates in this chapter are categorised and ordered as follows:

Structural: FeatureClassifier, StructuralFeatureClassifier, BehaviouralFeatureClassifier, Package.

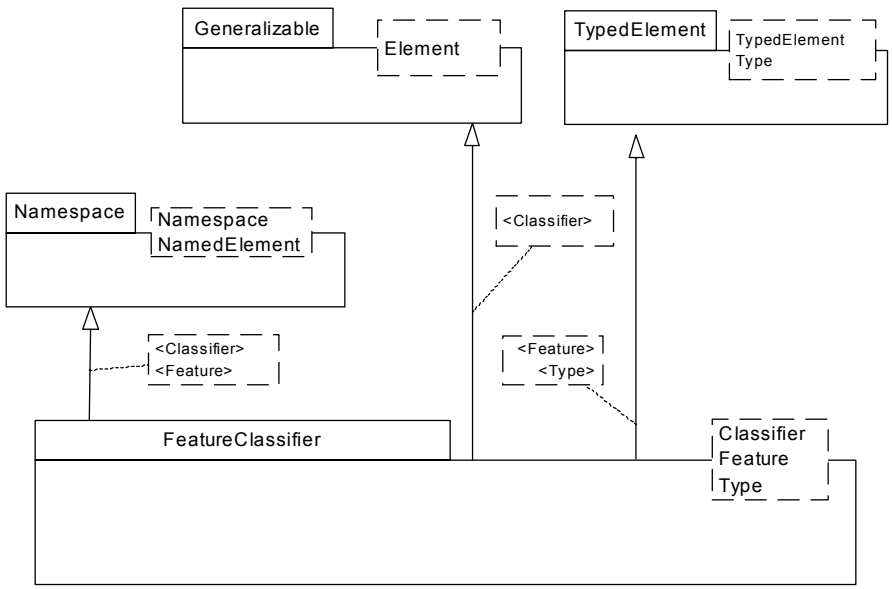
Semantics: StructuralFeatureClassifierValue, StructuralFeatureClassifierSemantics, BehaviouralFeatureClassifierValue, BehaviouralFeatureClassifierSemantics.

Extension: ExtendableNamespace, ExtendablePackage, ExtendableStructuralFeatureClassifier, ExtendableBehaviouralFeatureClassifier, TemplateInstantiation.

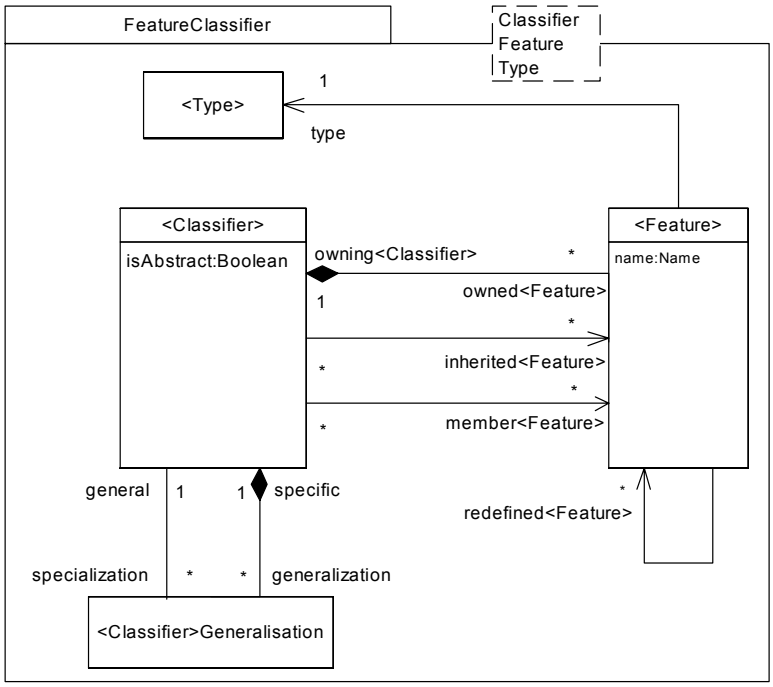
20.2 FEATURECLASSIFIER

Describes the general structure and properties of a classifier and its features.

20.2.1 Derivation



20.2.2 Definition



<Classifier>

Attributes

isAbstract True if the classifier is abstract

Associations

inherited<Feature> The inherited features of the classifier.

member<Feature> The features that are members of the namespace of the classifier.

owned<Feature> The features owned by the classifier.

specialization The specializations of the classifier.

generalization The generalizations of the classifier.

<Feature>

Attributes

name The name of the feature.

redefined<Feature> The features that are redefined.

type The type of the classifier.

Associations

owning<Classifier> The classifier that owns/contains the feature.

<Classifier>Generalization

Associations

general The general classifier.

specific The specific classifier.

20.2.3 Well-formedness Rules

<Classifier>

[1] The members of a classifier must include the owned features of the classifier.

```
context <Classifier> inv:
  self.member<Feature> ->includesAll(self.owned<Feature>)
```

[2] Circular inheritance is not permitted.

```
context <Classifier> inv:
  not self.allGeneralElements()->includes(self)
```

[3] Parent element's features must be inherited.

```
context <Classifier> inv:
  self.inherited<Feature> = self.generalElements()->iterate(p s = Set{} |
    s->union(p.member<Feature>->reject(x |
      self.member<Feature>->exists(x'|
        x'.redefined<Feature>->includes(x)))))
```

[4] Member features must include the inherited features.

```
context <Classifier> inv:
  self.member<Feature> ->includesAll(self.inherited<Feature>)
```

[5] Features cannot be owned and inherited.

```
context Class inv:
  self.owned<Feature>->intersection(self.inherited<Feature>) -> isEmpty
```

[6] Member features may only redefine parent features.

```
context <Classifier> inv:
  self.member<Feature> -> forAll(x |
    (self.generalElements() -> iterate(s = Set{} |
      s->union(g.member<Feature>)))->includesAll( x.redefined<Feature>)
```

<Feature>

[1] Redefined features must conform.

```
context <Feature> inv:
  self.redefined<Feature>->forAll(f |
    self.type.conformsTo(f.type))
```

20.2.4 Operations

<Classifier>

[1] Looks up a feature in a classifier given a name.

```
  context <Classifier>::lookup<Feature>forName(x:Name):featureClassifier::
    <Feature>
    self.member<Feature>->select(e|e.name = x ).selectElement()
```

[2] Looks up a name in a classifier given a feature.

```
context <Classifier>::lookupNameFor<Feature>(x : <Feature>): Name
    self.member<Feature>->select(e|e = x ).selectElement().name
```

[3] Returns the generalizations of the classifier.

```
context <Classifier>::generalElements() : Set(<Classifier>)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[4] Transitively returns all generalizations of the classifier.

```
context <Classifier>::allGeneralElements(): Set(<Classifier>)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

<Feature>

[1] Checks whether the supplied feature is in the same classifier as the feature.

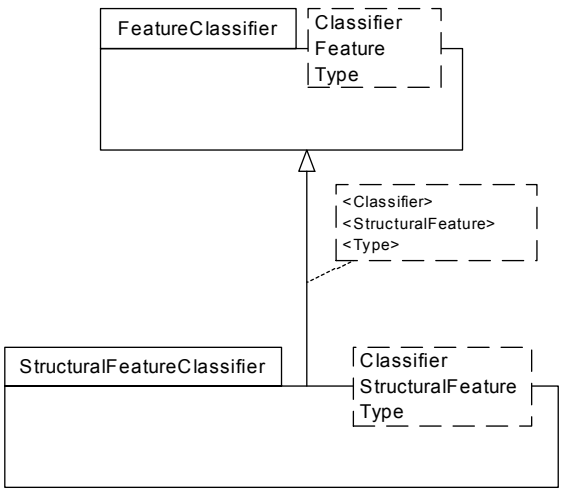
```
context <Feature>::sameNamespace(x : <Feature>) : Boolean
  x.slotValue(owning<Classifier>).member<Feature>->includes(self)
```

20.3 STRUCTURALFEATURECLASSIFIER

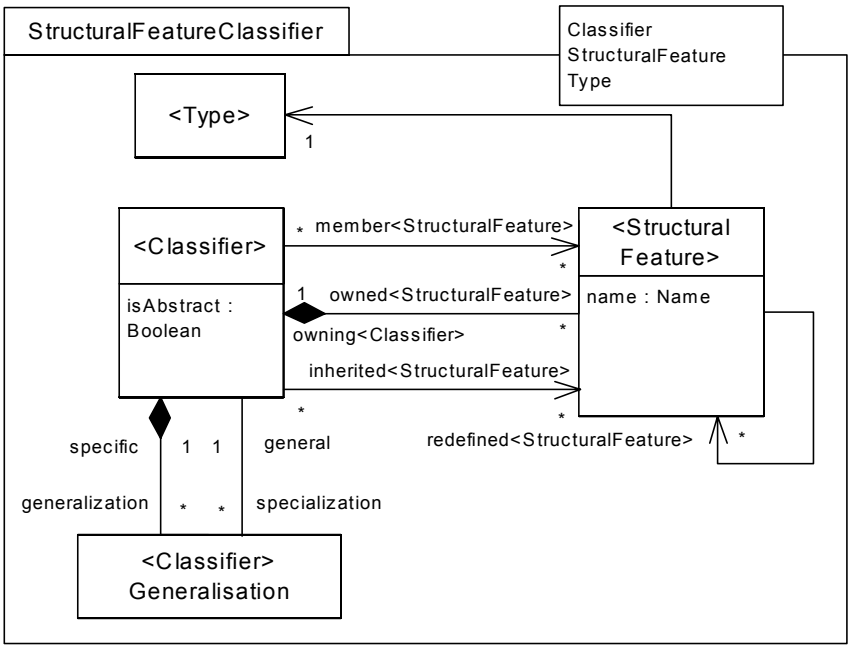
20.3.1 Summary

Describes the general structure and properties of a classifier and its structural features.

20.3.2 Derivation



20.3.3 Definition



<Classifier>

Attributes

isAbstract True if the classifier is abstract

Associations

inherited<Feature> The inherited structural features of the classifier.

member<Feature> The structural features that are members of the namespace of the classifier.

owned<Feature> The structural features owned by the classifier.

specialization The specializations of the classifier.

generalization The generalizations of the classifier.

<StructuralFeature>

Attributes

name The name of the structural feature.

redefined<Feature> The structural features that are redefined.

type The type of the classifier.

Associations

owning<Classifier> The classifier that owns/contains the structural feature.

<Classifier>Generalization

Associations

general The general classifier.

specific The specific classifier.

20.3.4 Well-formedness Rules

<Classifier>

[1] The members of a classifier must include the owned structural features of the classifier.

```
context <Classifier> inv:
  self.member<StructuralFeature> ->includesAll(self.owned<StructuralFeature>)
```

[2] Circular inheritance is not permitted.

```
context <Classifier> inv:
  not self.allGeneralElements()->includes(self)
```

[3] Parent structural features must be inherited.

```
context <Classifier> inv:
  self.inherited<StructuralFeature> = self.generalElements()->iterate(p
   s = Set{}|s->union(p.member<StructuralFeature>->reject(x |
      self.member<StructuralFeature>->exists(x'|
        x'.redefined<StructuralFeature>->includes(x)))))
```

[4] The member structural features must include the inherited structural features.

```
context <Classifier> inv:
  self.member<StructuralFeature> ->
    includesAll(self.inherited<StructuralFeature>)
```

[5] Structural features cannot be owned and inherited.

```
context Class inv:
  self.owned<StructuralFeature>->
    intersection(self.inherited<StructuralFeature>) -> isEmpty
```

[6] Member structural features must only redefine parent structural features.

```
context <Classifier> inv:
  self.member<StructuralFeature> -> forAll(x |
    (self.generalElements() -> iterate(s = Set{} |
      s->union(g.member<StructuralFeature>))))->includesAll
        ( x.redefined<StructuralFeature>)
```

<StructuralFeature>

[1] Redefined structural features must conform.

```
context <StructuralFeature> inv:
  self.redefined<StructuralFeature>->forAll(f |
    self.type.conformsTo(f.type))
```

20.3.5 Operations

<Classifier>

[1] Looks up a structural feature in a classifier given a name.

```
context <Classifier>::lookup<StructuralFeature>forName(x:Name):
                                                    <StructuralFeature>
    self.member<StructuralFeature>->select(e|e.name = x ).selectElement()
```

[2] Looks up a name in a classifier given a structural feature.

```
context<Classifier>::lookupNameFor<StructuralFeature>(x:<StructuralFeature>):
  Name
  self.member<StructuralFeature>->select(e|e = x ).selectElement().name
```

[3] Returns the generalizations of the classifier.

```
context <Classifier>::generalElements() : Set(<Classifier>)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[4] Transitively returns all generalizations of the classifier.

```
context <Classifier>::allGeneralElements(): Set(<Classifier>)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

<StructuralFeature>

[1] Checks whether the supplied structural feature is in the same classifier as the structural feature.

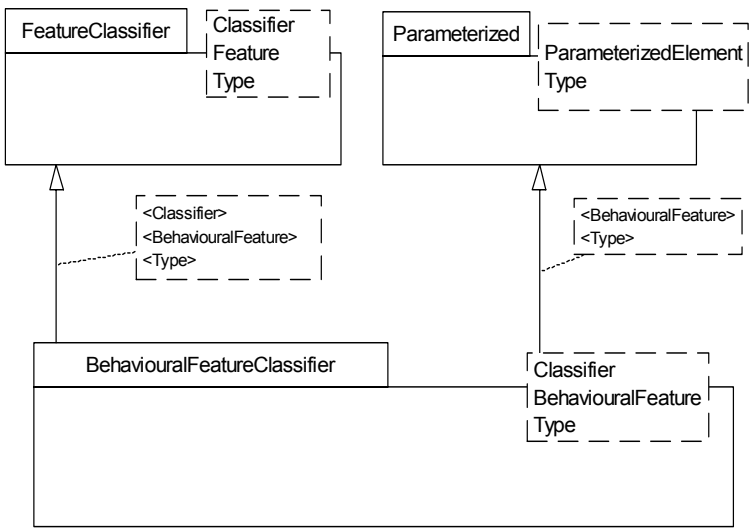
```
context <StructuralFeature>::sameNamespace(x : <StructuralFeature>) : Boolean
  x.owning<Classifier>.member<StructuralFeature>->includes(self)
```

20.4 BEHAVIOURALFEATURECLASSIFIER

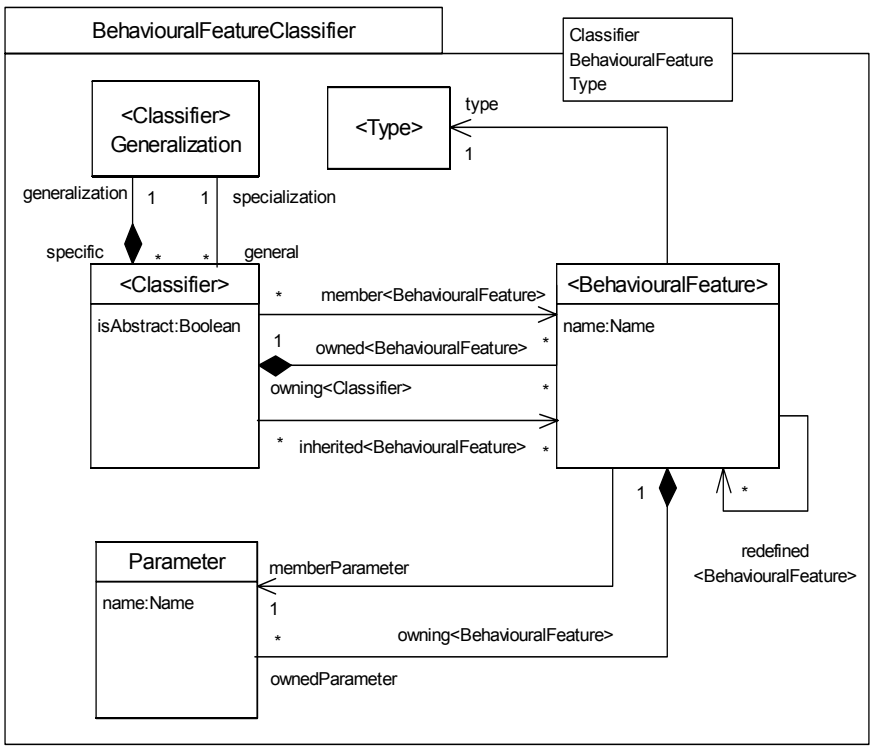
20.4.1 Summary

Describes the general structure and properties of a classifier and its behavioural features.

20.4.2 Derivation



20.4.3 Definition



<Classifier>

Attributes

isAbstract True if the classifier is abstract

Associations

inherited<BehaviouralFeature> The inherited behavioural features of the classifier.

member<BehaviouralFeature> The behavioural features that are members of the namespace of the classifier.

owned<BehaviouralFeature> The behavioural features owned by the classifier.

specialization The specializations of the classifier.

generalization The generalizations of the classifier.

<BehaviouralFeature>

Attributes

name The name of the behavioural feature.

redefined<BehaviouralFeature> The behavioural features that are redefined.

type The type of the classifier.

Associations

owning<Classifier> The classifier that owns/contains the behavioural feature.

<Classifier>Generalization

Associations

general The general Classifier.

specific The specific Classifier.

20.4.4 Well-formedness Rules

<Classifier>

[1] The members of a classifier must include the owned behavioural features of the classifier.

```
context <Classifier> inv:
  self.member<BehaviouralFeature> ->includesAll(self.owned<BehaviouralFeature>)
```

[2] Circular inheritance is not permitted.

```
context <Classifier> inv:
  not self.allGeneralElements()->includes(self)
```

[3] Parent behavioural features must be inherited.

```
context <Classifier> inv:
  self.inherited<BehaviouralFeature> = self.generalElements()->iterate(p
   s = Set{}|s->union(p.member<BehaviouralFeature>->reject(x |
      self.member<BehaviouralFeature>->exists(x'|
        x'.redefined<BehaviouralFeature>->includes(x)))))
```

[4] Member behavioural features must include the inherited behavioural features.

```
context <Classifier> inv:
  self.member<BehaviouralFeature>->
    includesAll(self.inherited<BehaviouralFeature>)
```

[5] Behavioural features cannot be owned and inherited.

```
context Class inv:
  self.owned<BehaviouralFeature>->
    intersection(self.inherited<BehaviouralFeature>) -> isEmpty
```

[6] Member behavioural features must only redefine parent behavioural features.

```
context <Classifier> inv:
  self.member<BehaviouralFeature> -> forAll(x |
    (self.generalElements() -> iterate(s = Set{} |
      s->union(g.member<BehaviouralFeature>))))-> includesAll
        (x.redefined<BehaviouralFeature>)
```

<BehaviouralFeature>

[1] Redefined behavioural features must conform.

```
context <BehaviouralFeature> inv:
  self.redefined<BehaviouralFeature>->forAll(f |
    self.type.conformsTo(f.type))
```

[2] The type of the parameter of the behavioural feature must conform to its parent's type.

```
context <BehaviouralFeature> inv:
  self.redefined<BehaviouralFeature> -> forAll(f |
   (1).to(self.parameter->size) -> forAll(n |
      self.parameter.at(n).type.conformsTo(f.parameter.at(n).type)))
```

20.4.5 Operations

<Classifier>

[1] Looks up the behavioural feature in a classifier given a name.

```
context <Classifier>::lookup<BehaviouralFeature>forName(x:Name):
                                                    <BehaviouralFeature>
  self.member<BehaviouralFeature>->select(e|e.name = x ).selectElement()
```

[2] Looks up the name in a classifier given a behavioural feature.

```
context <Classifier>::lookupNameFor<BehaviouralFeature>(x :
 <BehaviouralFeature>): Name
 self.member<BehaviouralFeature>->select(e|e = x ).selectElement().name
```

[3] Returns the generalizations of the classifier.

```
context <Classifier>::generalElements() : Set(<Classifier>)
  self.generalization->iterate(p s=Set{} | s->union(Set{p.general}))
```

[4] Transitively returns all generalizations of the classifier.

```
context <Classifier>::allGeneralElements(): Set(<Classifier>)
  self.generalElements()->iterate(g s=self.generalElements() |
    s->union(g.allGeneralElements()))
```

<BehaviouralFeature>

[1] Checks whether the given behavioural feature is in the same classifier.

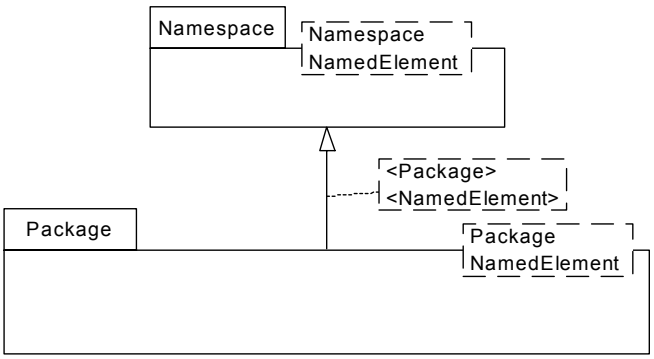
```
context <BehaviouralFeature>::sameNamespace(x:<BehaviouralFeature>):Boolean
  x.owning<Classifier>.member<BehaviouralFeature>->includes(self)
```

20.5 PACKAGE

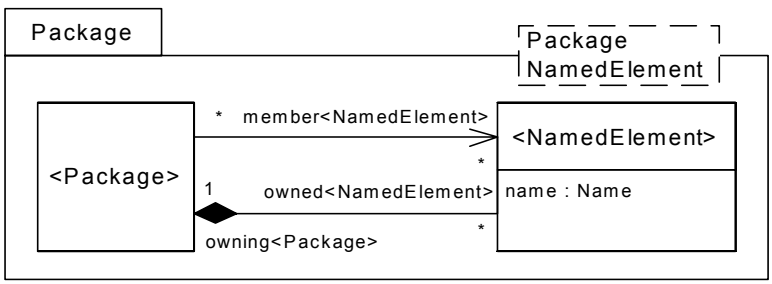
20.5.1 Summary

Defines a package template. A package is a large grained module structure that contains named elements..

20.5.2 Derivation



20.5.3 Definition



<Package>

Attributes

member<NamedElement> The named elements that are members of the package namespace.

Associations

owned<NamedElement> The owned named elements.

<NamedElement>

Attributes

name The name of the named element.

Associations

owningPackage The package that owns the named element.

20.5.4 Well-formedness Rules

<Package>

[1] The members of a package must include the owned elements of the package.

```
context <Package> inv:
  self.member<NamedElement> ->includesAll(self.owned<NamedElement>)
```

[2] No two elements must have the same name in the package.

```
context <Package> inv:
  self.member<NamedElement> -> forAll(e1 |
    self.member<NamedElement> -> forAll(e2 |
      e1 <> e2 implies e1.name <> e2.name))
```

20.5.5 Operations

<Package>

[1] Looks up the named element in a package given a name.

```
context <Package>::lookup<NamedElement>forName(x : Name): <NamedElement>
  self.member<NamedElement>->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a package given a named element.

```
context <Package>::lookupNameFor<NamedElement>(n : <NamedElement>):Name
  self.member<NamedElement>->select(e | e = x).selectElement().name
```

<NamedElement>

[1] Checks whether the supplied named element is in the same package as the named element.

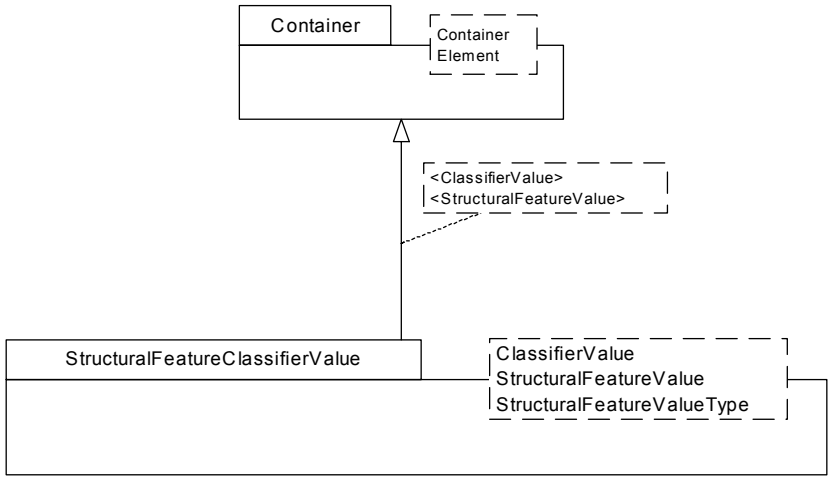
```
context <NamedElement>::samePackage(x : <NamedElement>):Boolean
  x.owning<Package>.member<NamedElement> -> includes(self)
```

20.6 STRUCTUALFEATURECLASSIFIERVALUE

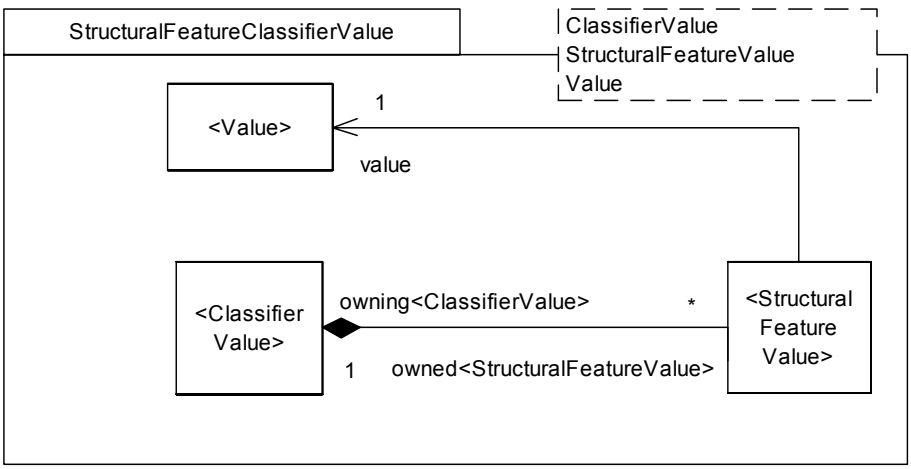
20.6.1 Summary

Describes the values of classifiers with structural features.

20.6.2 Derivation



20.6.3 Definition



<ClassifierValue>

Associations

owned<StructuralFeatureValue> The set of structural feature values owned by the classifier.

<StructuralFeatureValue>

Associations

value The value of the structural feature value.

20.6.4 Well-formedness Rules

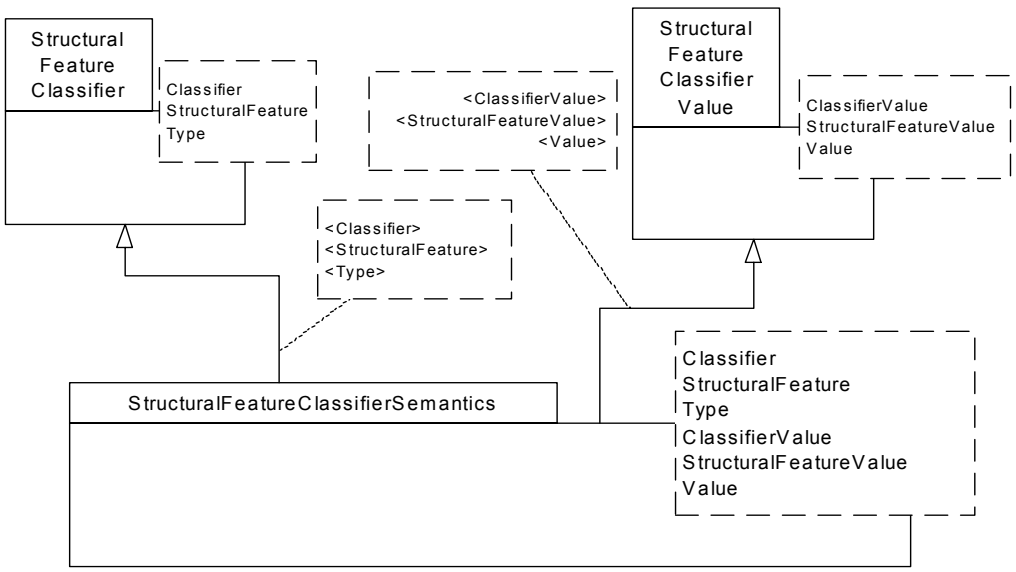
20.6.5 Operations

20.7 STRUCTURALFEATURECLASSIFIERSEMANTICS

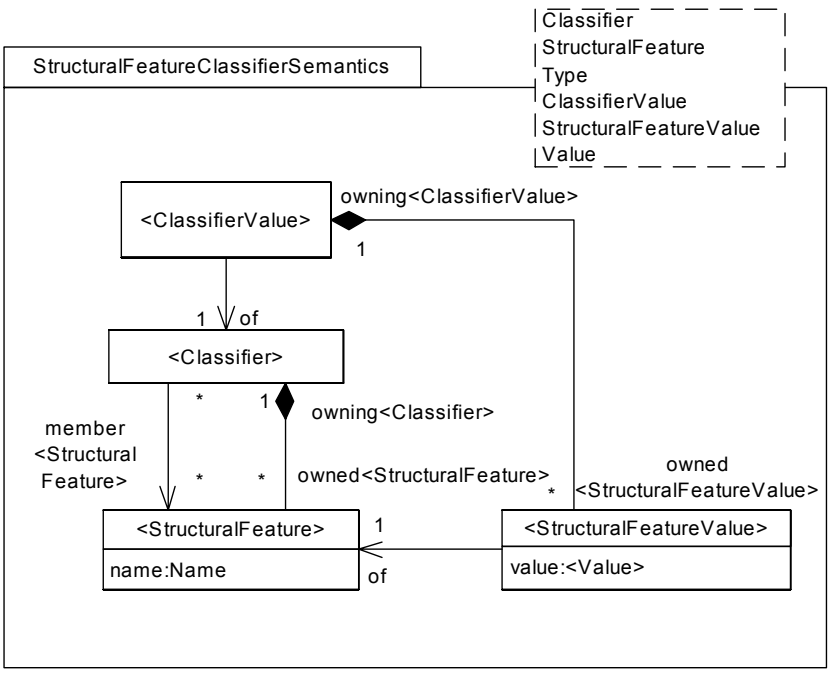
20.7.1 Summary

Defines the semantics for structural features of a classifier.

20.7.2 Derivation



20.7.3 Definition



<Classifier>

Associations

owned<*StructuralFeature*> The owned structural features of the classifier.

member<*StructuralFeature*> The member structural features of the classifier.

<ClassifierValue>

Attributes

of The classifier that this is a value of.

<StructuralFeature>

Attributes

name The name of the structural feature.

Associations

owning<*Classifier*> The classifier that owns the feature.

<StructuralFeatureValue>

Attributes

of The structural feature that this is a value of.

Associations

owning<*ClassifierValue*> The owning classifier value.

20.7.4 Well-formedness Rules

<ClassifierValue>

[1] All values of classifier contain values of its structural features.

```
context <ClassifierValue> inv:
  self.of.member<StructuralFeature> -> forAll(c |
    self.owned<StructuralFeatureValue> -> exists(d | d.of = c))
```

[2] All contained structural feature values must be values of some structural feature in the classifier.

```
context <ClassifierValue> inv:
  self.owned<StructuralFeatureValue> -> forAll(c |
    self.of.member<StructuralFeature> -> exists(d | c.of = d))
```

<StructuralFeatureValue>

[1] The type of the value of the structural feature value must conform to the type of its structural feature.

```
context <StructuralFeatureValue> inv:
  self.value.of.conformsTo(self.of.type)
```

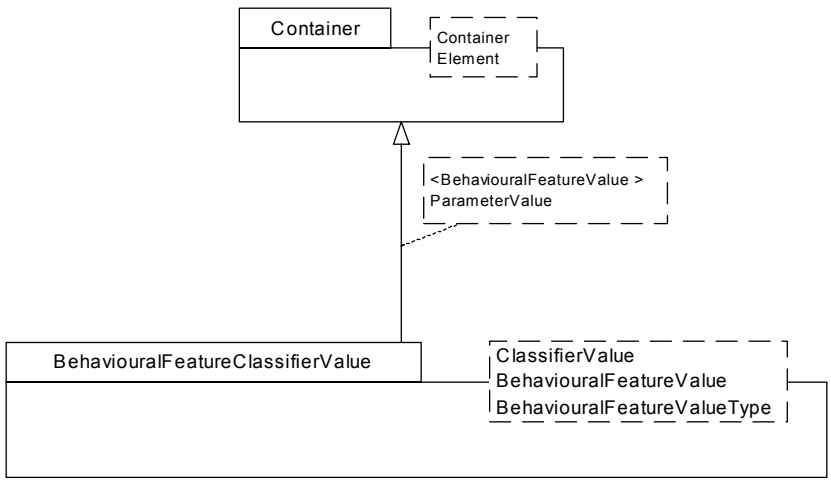
20.7.5 Operations

20.8 BEHAVIOURALFEATURECLASSIFIERVALUE

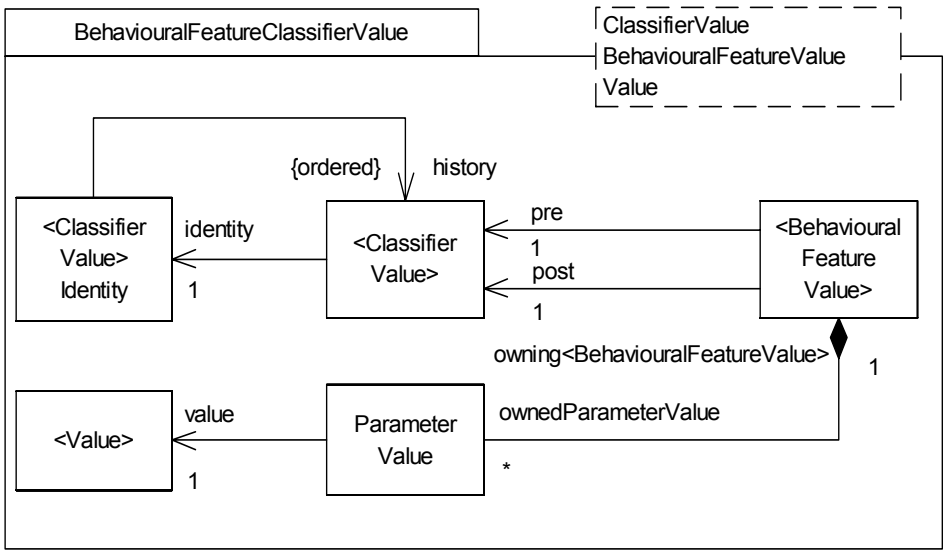
20.8.1 Summary

Describes the values of classifiers with behavioural features.

20.8.2 Derivation



20.8.3 Definition



<BehaviouralFeatureValue>

Associations

ownedParameterValue The owned parameter values.

pre The pre value of the behavioural feature value.

post The pre value of the behavioural feature value.

<ParameterValue>

Associations

value The value of the parameter value.

20.8.4 Well-formedness Rules

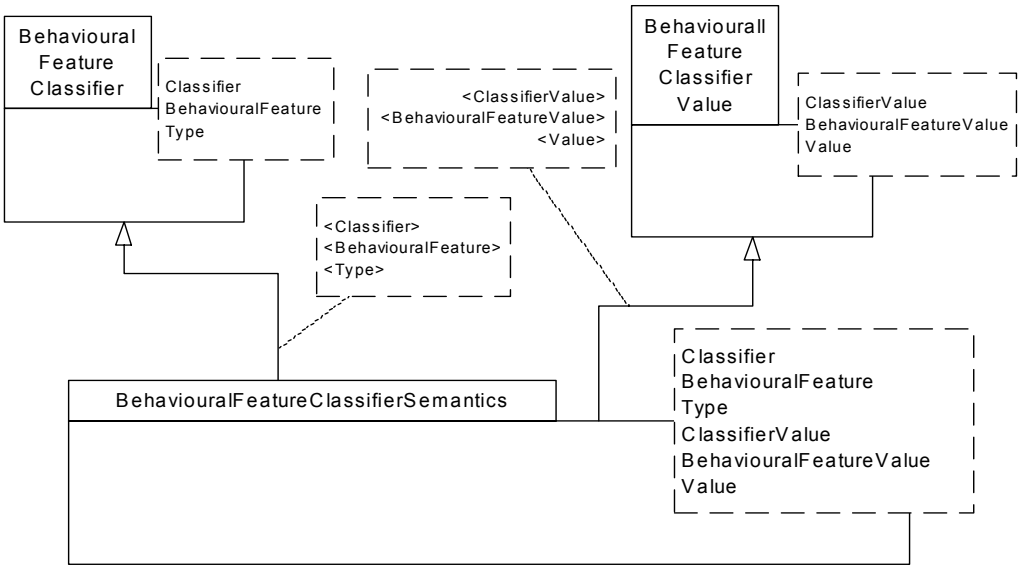
20.8.5 Operations

20.9 BEHAVIOURALFEATURECLASSIFIERSEMANTICS

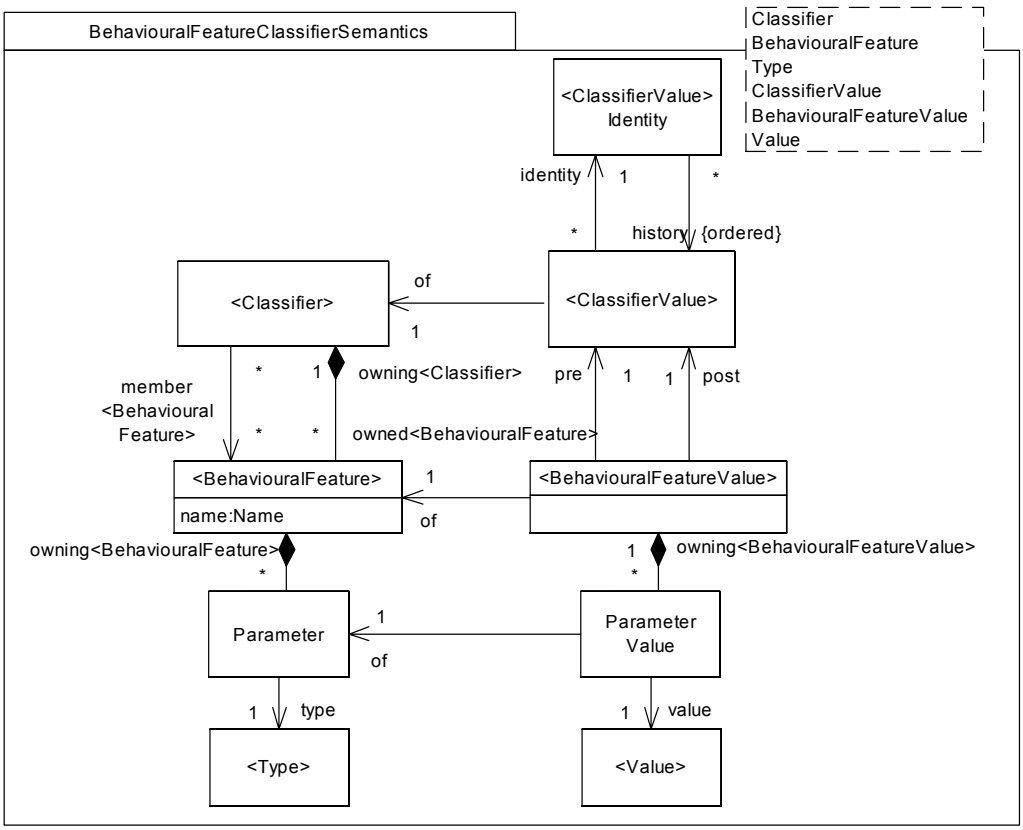
20.9.1 Summary

Defines the semantics for behavioural features of a classifier.

20.9.2 Derivation



20.9.3 Definition



<Classifier>

Associations

owned<BehaviouralFeature> The owned behavioural features of the classifier.

member<BehaviouralFeature> The behavioural features that are members of the classifier namespace.

<ClassifierValue>

Associations

of The classifier that this is a value of.

identity The identity of the classifier value.

<BehaviouralFeature>

Attributes

name The name of the behavioural feature.

Associations

owning<Classifier> The classifier that owns the feature.

<BehaviouralFeatureValue>

Attributes

of The behavioural feature that this is a value of.

Associations

owning<ClassifierValue> The owning classifier value.

20.9.4 Well-formedness Rules

<ClassifierValue>

[1] All values of classifier contain values of its structural features.

```
context <ClassifierValue> inv:
  self.of.member<BehaviouralFeature> -> forAll(c |
    self.owned<BehaviouralFeatureValue> -> exists(d | d.of = c))
```

[2] All contained structural feature values must be values of some structural feature in the classifier.

```
context <ClassifierValue> inv:
  self.owned<BehaviouralFeatureValue> -> forAll(c |
    self.of.member<BehaviouralFeature> -> exists(d | c.of = d))
```

<StructuralFeatureValue>

[1] The type of the value of the structural feature value must conform to the type of its structural feature.

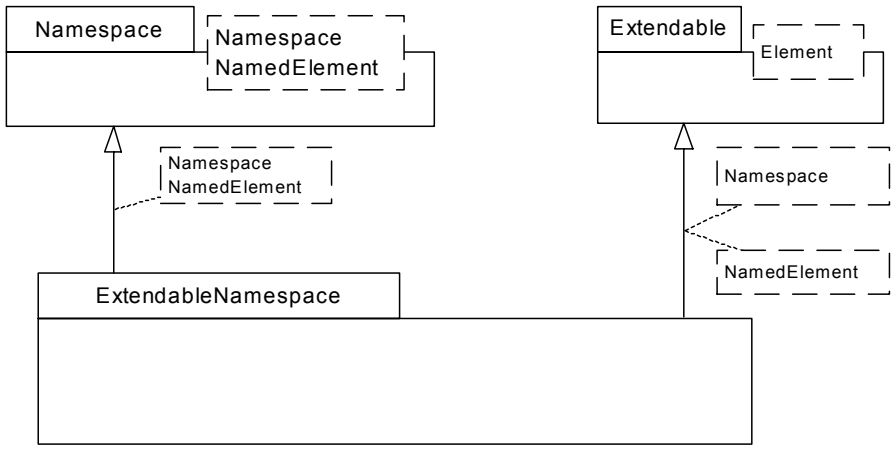
```
context <StructuralFeatureValue> inv:
  self.value.of.conformsTo(self.of.type)
```

20.10 EXTENDABLENAMESPACE

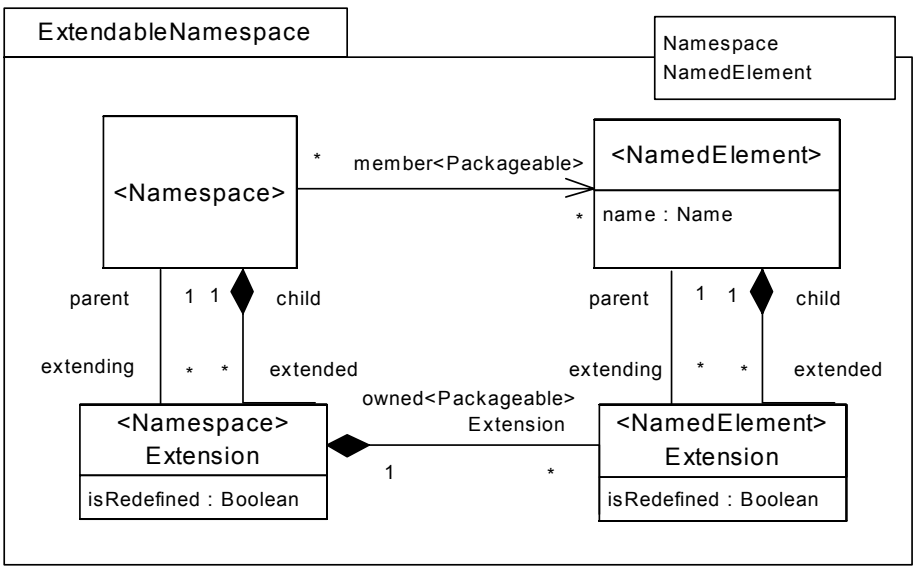
20.10.1 Summary

An extension relationship between namespaces. When a namespace extends another namespace, the members of the parent namespace are extended into the namespace of the child namespace.

20.10.2 Derivation



20.10.3 Definition



<Namespace>

Attributes

member<NamedElement> The members of the namespace

Associations

extended The extended namespaces.

extending The extending namespaces.

owned<NamedElement> The owned named elements.

<NamedElement>

Attributes

name The name of the named element.

Associations

owningNamespace The namespace owning the named element.

extended The extended named elements.

extending The extending named elements.

<Namespace>Extension

Attributes

isRedefined True if the extension is redefined.

Associations

parent The parent namespace.

child The child namespace.

owned<NamedElement>Extension The owned named element extensions.

<NamedElement>Extension

Associations

parent The parent named element.

child The child named element.

20.10.4 Well-formedness Rules

<Namespace>

[1] The members of a namespace must include its inherited elements.

```
context <Namespace> inv:
  self.member<NamedElement>.includesAll(self.inherited<NamedElement>)
```

[2] The members of a namespace must include its owned elements.

```
context <Namespace> inv:
  self.member<NamedElement>.includesAll(self.owned<NamedElement>)
```

[3] Circular inheritance is not permitted.

```
context <Namespace> inv:
  not self.allExtendedElements()->includes(self)
```

<NamedElement>

[3] Circular inheritance is not permitted

```
context <NamedElement> inv:
  not self.allExtendedElements()->includes(self)
```

<Namespace>Extension

[1] The members of the parent namespace are extended into the namespace of the child namespace.

```
context <Namespace>Extension inv:
  self.parent.member<NamedElement>->forAll(e |
    self.owned<NamedElement>Extension->exists(e' |
      e'.parent = e and
        self.child.member<NamedElement>->exists(e'' |
          e'.child = e'')))
```

20.10.5 Operations

<Namespace>

[1] Looks up a named element in a namespace given a name.

```
context <Namespace>::lookup<NamedElement>forName(x : Name): <NamedElement>
  self.member<NamedElement>->select(e | e.name = x).selectElement()
```

[2] Looks up a name in a namespace given a named element.

```
context <Namespace>::lookupNameFor<NamedElement>(n : <NamedElement>):Name
  self.member<NamedElement>->select(e | e = x).selectElement().name
```

[3] Returns the namespaces that have been extended.

```
context <Namespace>::extendedElements():Set(<Namespace>)
  self.extended -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[4] Transitively returns all namespaces that have been extended.

```
context Namespace::allExtendingElements():Set(Namespace)
  self.extendedElements()->iterate(g s = self.extendedElements() |
    s->union(g.allExtendingElements()))
```

<NamedElement>

[1] Checks whether the given named element is owned by the same name space as this named element.

```
context <NamedElement>::sameNamespace(x : <NamedElement>):Boolean
  x.owning<Namespace>.member<NamedElement> -> includes(self)
```

[2] Returns the named elements that have been extended.

```
context <NamedElement>::extendingElements():Set(<NamedElement>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[3] Transitively returns all named elements that have extended.

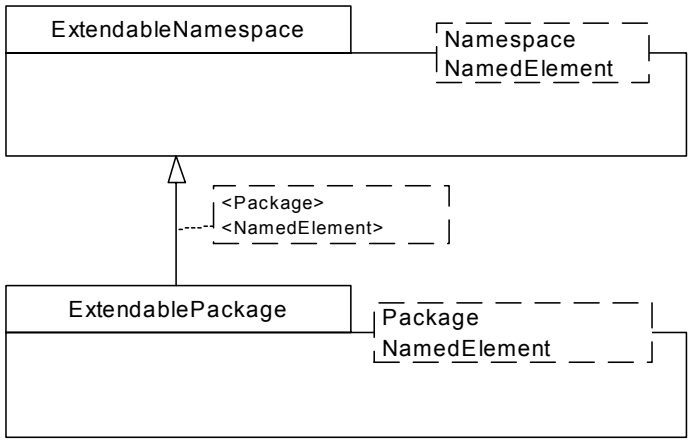
```
context NamedElement::allExtendingElements():Set(NamedElement)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

20.11 EXTENDABLEPACKAGE

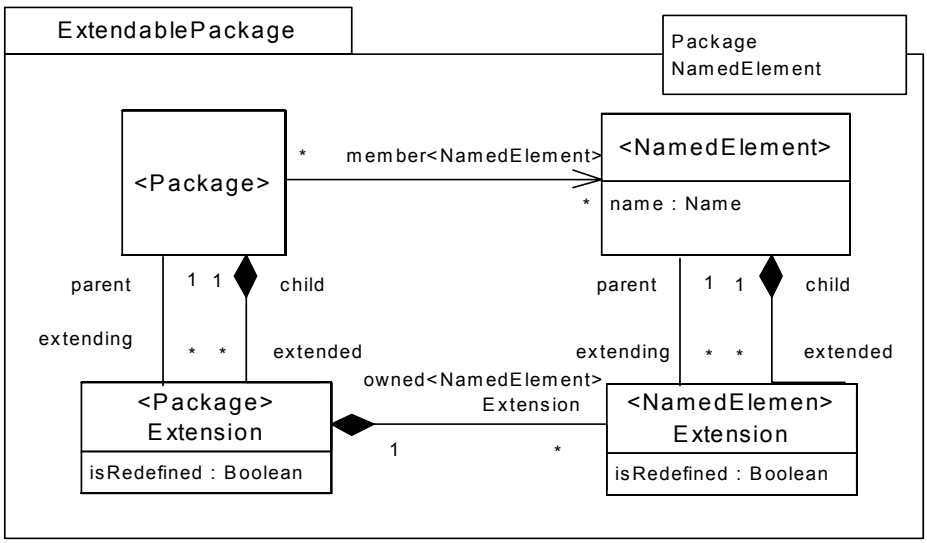
20.11.1 Summary

This templates defines a package that can be extended.

20.11.2 Derivation



20.11.3 Definition



<Package>

Associations

extended The extended elements of the Package.

extending The extending elements of the Package.

owned<NamedElement> The owned named elements.

member<NamedElement> The member named elements belonging to the Package namespace.

inherited<NamedElement> The inherited named elements.

<NamedElement>

Attributes

name The name of the NamedElement

Associations

owningPackage The Package owning this NamedElement.

extended The extended elements of the NamedElement.

extending The extending elements of the NamedElement.

<Package>Extension

Attributes

isRedefined True if the extension is a redefinition.

Associations

parent The parent <Package> in the pair of <Package>s it links.

child The child <Package> of the pair of <Package>s it links.

owned<NamedElement>Extension The set of <Named>Extensions owned.

<NamedElement>Extension

Associations

parent The parent NamedElement.

child The child NamedElement.

20.11.4 Well-formedness Rules

<Package>

[1] The members of the package must include its inherited named elements.

```
context <Package> inv:
  self.member<NamedElement>-> includesAll
    (self.inherited<NamedElement>)
```

[2] The members of a Package must include the owned named elements of the Package.

```
context <Package> inv:
  self.member<NamedElement>->includesAll(self.owned<NamedElement>)
```

[3] Circular inheritance is not permitted.

```
context <Package> inv:
  not self.allExtendingElements()->includes(self)
```

<NamedElement>

[3] Circular inheritance is not permitted.

```
context <NamedElement> inv:
  not self.allExtendingElements()->includes(self)
```

<Package>Extension

[1] Parent's elements must be extended into the namespace of the child.

```
context <Package>Extension inv:
  self.parent.member<NamedElement>->forAll(e |
    self.owned<NamedElement>Extension->exists(e' |
      e'.parent = e and
        self.child.member<NamedElement>->exists(e'' |
          e'.child = e'')))
```

[2] If the child doesn't equal the parent in an owned named element extension then it must be owned by the child.

```
context <Package>Extension inv:
  self.owned<NamedElement>Extension -> forAll(e |
    e.child <> e.parent implies
      self.child.owned<NamedElement> -> includes(e.child))
```

20.11.5 Operations

<Package>

[1] Looks up the NamedElement in a Package given a name.

```
context <Package>::lookup<NamedElement>forName(x : Name): <NamedElement>
  self.member<NamedElement>->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a Package given a NamedElement.

```
context <Package>::lookupNameFor<NamedElement>(n : <NamedElement>):Name
  self.member<NamedElement>->select(e | e = x).selectElement().name
```

[3] Returns the packages it has extended from.

```
context <Package>::extendingElements():Set(<Package>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[4] Transitively returns the set of all named elements it has extended from.

```
context Package::allExtendingElements():Set(<Package>)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

<NamedElement>

[1] Checks whether the given NamedElement is in the same Package.

```
context <NamedElement>::sameNamespace(x : <NamedElement>):Boolean
  x.owning<Package>.member<NamedElement> -> includes(self)
```

[2] Returns the named elements it has extended from.

```
context <NamedElement>::extendingElements():Set(<NamedElement>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[3] Transitively returns the set of all named elements it has extended from.

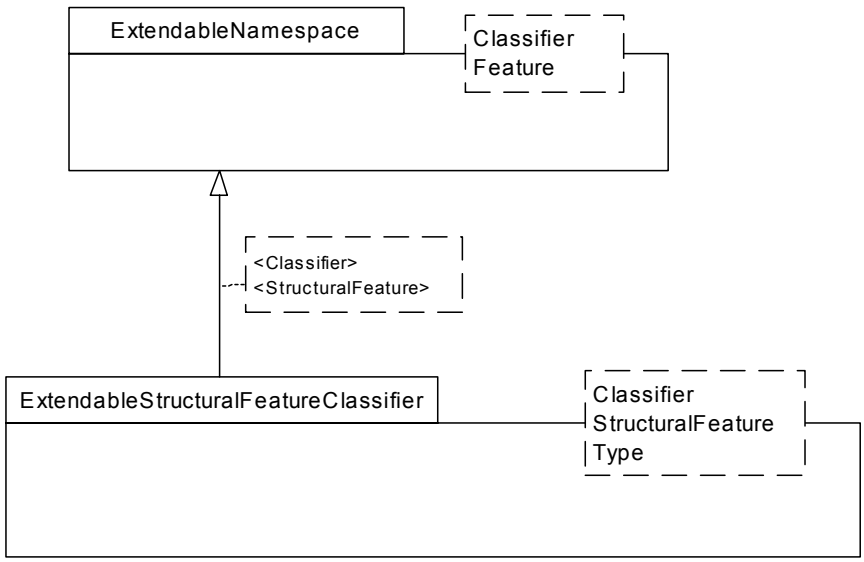
```
context NamedElement::allExtendingElements():Set(NamedElement)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

20.12 EXTENDABLESTRUCTURALFEATURECLASSIFIER

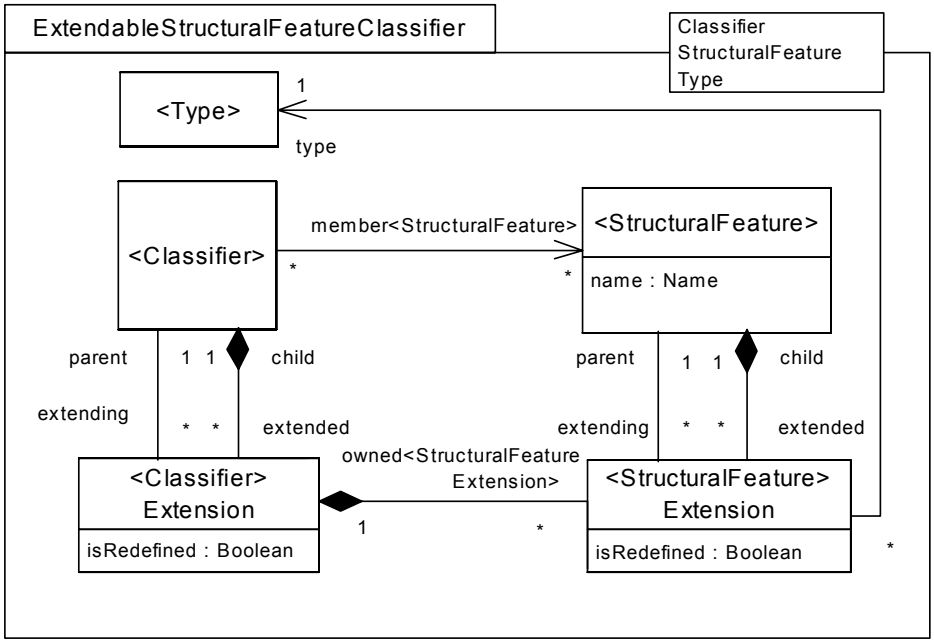
20.12.1 Summary

This template defines the structural features of a classifier that can be extended.

20.12.2 Derivation



20.12.3 Definition



<Classifier>

Attributes

member<StructuralFeature> The structural features that are members of the namespace of the Classifier.

inherited<StructuralFeature> The inherited structural features.

Associations

extended The extended classifiers.

extending The extending classifiers.

owned<StructuralFeature> The owned structural features.

<StructuralFeature>

Attributes

name The name of the StructuralFeature.

Associations

owningClassifier The Classifier owning this StructuralFeature.

extended The extended structural features.

extending The extending structural features.

<Classifier>Extension

Attributes

isRedefined True if the extension is redefined.

Associations

parent The parent Classifier.

child The child Classifier.

owned<StructuralFeature>Extension The owned extensions.

<StructuralFeature>Extension

Associations

parent The parent StructuralFeature.

child The child StructuralFeature.

20.12.4 Well-formedness Rules

<Classifier>

[1] The member structural features must include the inherited structural features.

```
context <Classifier> inv:
  self.member<StructuralFeature>-> includesAll
    (self.inherited<StructuralFeature>)
```

[2] The member structural features of a Classifier must include the owned structural features of the Classifier.

```
context <Classifier> inv:
  self.member<StructuralFeature>->includesAll(self.owned<StructuralFeature>)
```

[3] Circular inheritance is not permitted.

```
context <Classifier> inv:
  not self.allExtendingElements()->includes(self)
```

<StructuralFeature>

[1] Circular inheritance is not permitted.

```
context <StructuralFeature> inv:
  not self.allExtendingElements()->includes(self)
```

<Classifier>Extension

[1] Parent's structural features must be extended into the namespace of the child.

```
context <Classifier>Extension inv:
  self.parent.member<StructuralFeature>->forAll(e |
    self.owned<StructuralFeature>Extension->exists(e' |
      e'.parent = e and
        self.child.member<StructuralFeature>->exists(e'' |
          e'.child = e'')))
```

[2] If the child doesn't equal the parent in an owned structural feature extension then it must be owned by the child.

```
context <Classifier>Extension inv:
  self.owned<StructuralFeature>Extension -> forAll(e |
    e.child <> e.parent implies
      self.child.owned<StructuralFeature> -> includes(e.child))
```

<StructuralFeature>Extension

[1] This conformsTo relationship is similar to conformsTo, however, it must check that if the types are classes then the child *extends* the parent.

```
context <StructuralFeature>Extension inv:
  self.child.type.conformsToExtension(self.parent.type)
```

[1] If an extension has occurred (as opposed to inheritance) then the type of the child StructuralFeature should be in the same namespace as the child StructuralFeature's classifier.

```
context <StructuralFeature>Extension inv:
  self.child <> self.parent implies
    self.child.owning<Classifier>.sameNamespace(self.child.type)
```

20.12.5 Operations

<Classifier>

[1] Looks up the StructuralFeature in a Classifier given a name.

```
context <Classifier>::lookup<StructuralFeature>forName(x : Name):
  <StructuralFeature>
  self.member<StructuralFeature>->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a Classifier given a StructuralFeature.

```
context <Classifier>::lookupNameFor<StructuralFeature>(n :
  <StructuralFeature>):Name
  self.member<StructuralFeature>->select(e | e = x).selectElement().name
```

[3] Returns the set of classifiers it has extended from.

```
context <Classifier>::extendingElements():Set(<Classifier>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[4] Transitively returns the set of all classifiers it has extended from.

```
context Classifier::allExtendingElements():Set(Classifier)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

<StructuralFeature>

[1] Checks whether the given StructuralFeature is in the same Classifier.

```
context <StructuralFeature>::sameNamespace(x : <StructuralFeature>):Boolean
  x.owning<Classifier>.member<StructuralFeature> -> includes(self)
```

[2] Returns the set of structural features it has extended from.

```
context <StructuralFeature>::extendingElements():Set(<StructuralFeature>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[3] Transitively returns the set of all structural features it has extended from.

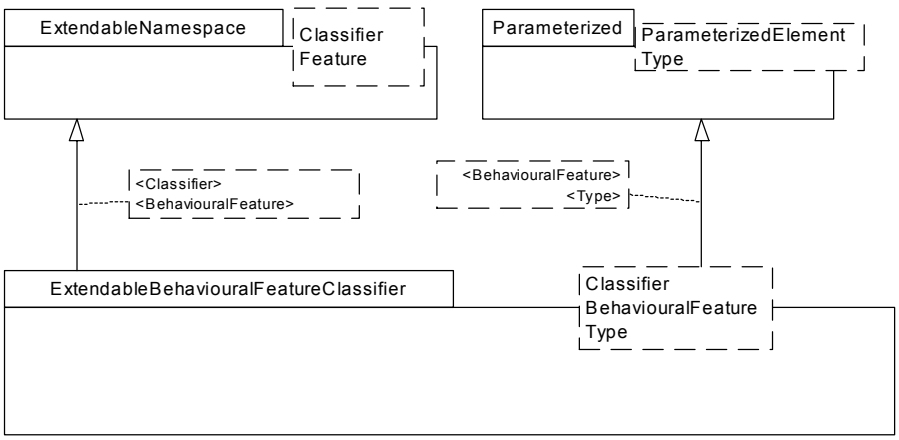
```
context StructuralFeature::allExtendingElements():Set(StructuralFeature)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

20.13 EXTENDABLEBEHAVIOURALFEATURECLASSIFIER

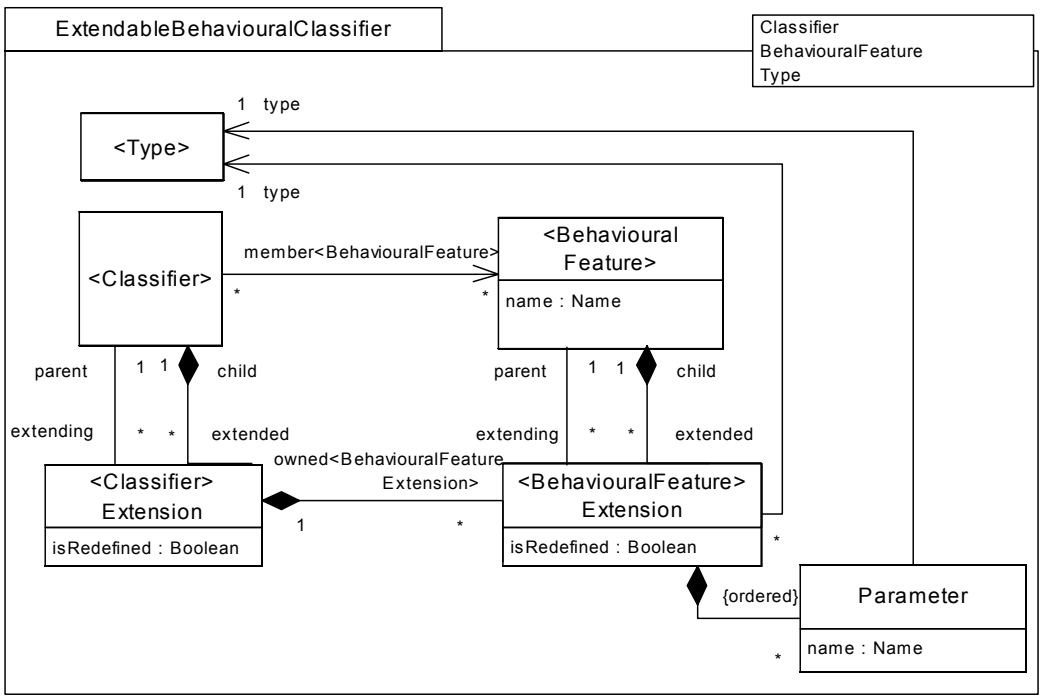
20.13.1 Summary

This template defines the behavioural features of a classifier that can be extended.

20.13.2 Derivation



20.13.3 Definition



<Classifier>

Attributes

member<BehaviouralFeature> The behavioural features that are members of the namespace of the Classifier.

inherited<BehaviouralFeature> The inherited behavioural features.

Associations

extended The extended classifiers.

*extending*The extending classifiers.

owned<BehaviouralFeature> The owned behavioural features.

<BehaviouralFeature>

Attributes

name The name of the BehaviouralFeature.

member<Parameter> The parameters of the behavioural feature's namespace.

Associations

owningClassifier The Classifier owning this BehaviouralFeature.

owned<Parameter> The owned parameters.

extended The extended behavioural features.

extending The extending behavioural feature.

<Parameter>

Attributes

name The name of the Parameter

Associations

owningBehaviouralFeature The BehaviouralFeature owning this Parameter.

<Classifier>Extension

Attributes

isRedefined True if the extension is redefined.

Associations

parent The parent Classifier.

child The child Classifier.

owned<BehaviouralFeature>Extension The owned behavioural feature extensions.

<BehaviouralFeature>Extension

AssociationsAttributes

parent The parent BehaviouralFeature.

child The child BehaviouralFeature.

20.13.4 Well-formedness Rules

<Classifier>

[1] The members of the Classifier must include the inherited elements.

```
context <Classifier> inv:
  self.member<BehaviouralFeature> -> includesAll
    (self.inherited<BehaviouralFeature>)
```

[2] The members of a Classifier must include the owned elements of the Classifier.

```
context <Classifier> inv:
  self.member<BehaviouralFeature>->includesAll(self.owned<BehaviouralFeature>)
```

[3] Circular inheritance is not permitted.

```
context <Classifier> inv:
  not self.allExtendingElements()->includes(self)
```

<BehaviouralFeature>

[1] Circular inheritance is not permitted.

```
context <BehaviouralFeature> inv:
  not self.allExtendingElements()->includes(self)
```

[2] The parameters of a Behavioural Feature must include its owned parameters.

```
context <BehaviouralFeature> inv:
  self.member<Parameter> ->includesAll(self.owned<Parameter>)
```

<Classifier>Extension

[1] Parent's elements must be preserved.

```
context <Classifier>Extension inv:
  self.parent.member<BehaviouralFeature>->forAll(e |
    self.owned<BehaviouralFeature>Extension->exists(e' |
      e'.parent = e and
        self.child.member<BehaviouralFeature>->exists(e'' |
          e'.child = e'')))
```

[2] If the child doesn't equal the parent in an owned behavioural feature extension then it must be owned by the child.

```
context <Classifier>Extension inv:
  self.owned<BehaviouralFeature>Extension -> forAll(e |
    e.child <> e.parent implies
      self.child.owned<BehaviouralFeature> -> includes(e.child))
```

<BehaviouralFeature>Extension

[1] This conformsTo relationship is similar to conformsTo, however, it must check that if the types are classes then the child *extends* the parent.

```
context <BehaviouralFeature>Extension inv:
  self.child.type.conformsToExtension(self.parent.type)
```

[1] If an extension has occurred (as opposed to inheritance) then the type of the child BehaviouralFeature should be in the same namespace as the child BehaviouralFeature's classifier.

```
context <BehaviouralFeature>Extension inv:
  self.child <> self.parent implies
    self.child.owning<Classifier>.sameNamespace(self.child.type)
```

20.13.5 Operations

<Classifier>

[1] Looks up the BehaviouralFeature in a Classifier given a name.

```
context <Classifier>::lookup<BehaviouralFeature>forName(x : Name):
  <BehaviouralFeature>
  self.member<BehaviouralFeature>->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a Classifier given a BehaviouralFeature.

```
context <Classifier>::lookupNameFor<BehaviouralFeature>(n :
  <BehaviouralFeature>):Name
  self.member<BehaviouralFeature>->select(e | e = x).selectElement().name
```

[3] Returns the set of classifiers it has extended from.

```
context <Classifier>::extendingElements():Set(<Classifier>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[4] Transitively returns the set of all classifiers it has extended from.

```
context Classifier::allExtendingElements():Set(Classifier)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

<BehaviouralFeature>

[1] Looks up the Parameter in a `BehaviouralFeature` given a name.

```
context <BehaviouralFeature>::lookup<Parameter>forName(x : Name): <Parameter>
  self.member<Parameter>->select(e | e.name = x).selectElement()
```

[2] Looks up the name in a Classifier given a Parameter.

```
context <BehaviouralFeature>::lookupNameFor<Parameter>(n : <Parameter>):Name
  self.member<Parameter>->select(e | e = x).selectElement().name
```

[3] Checks whether the given BehaviouralFeature is in the same Classifier.

```
context <BehaviouralFeature>::sameNamespace(x : <BehaviouralFeature>):Boolean
  x.owning<Classifier>.member<BehaviouralFeature> -> includes(self)
```

[4] Returns the set of behavioural features it has extended from.

```
context <BehaviouralFeature>::extendingElements():Set(<BehaviouralFeature>)
  self.extending -> iterate(p s = Set{} | s->union(Set{p.parent}))
```

[5] Transitively returns the set of all behavioural features it has extended from.

```
context BehaviouralFeature::allExtendingElements():Set(BehaviouralFeature)
  self.extendingElements()->iterate(g s = self.extendingElements() |
    s->union(g.allExtendingElements()))
```

<Parameter>

[1] Checks whether the given Parameter is in the same BehaviouralFeature.

```
context <Parameter>::sameNamespace(x : <Parameter>):Boolean
  x.<owning<Classifier>.member<Parameter> -> includes(self)
```

20.14 TEMPLATEINSTANTIATION

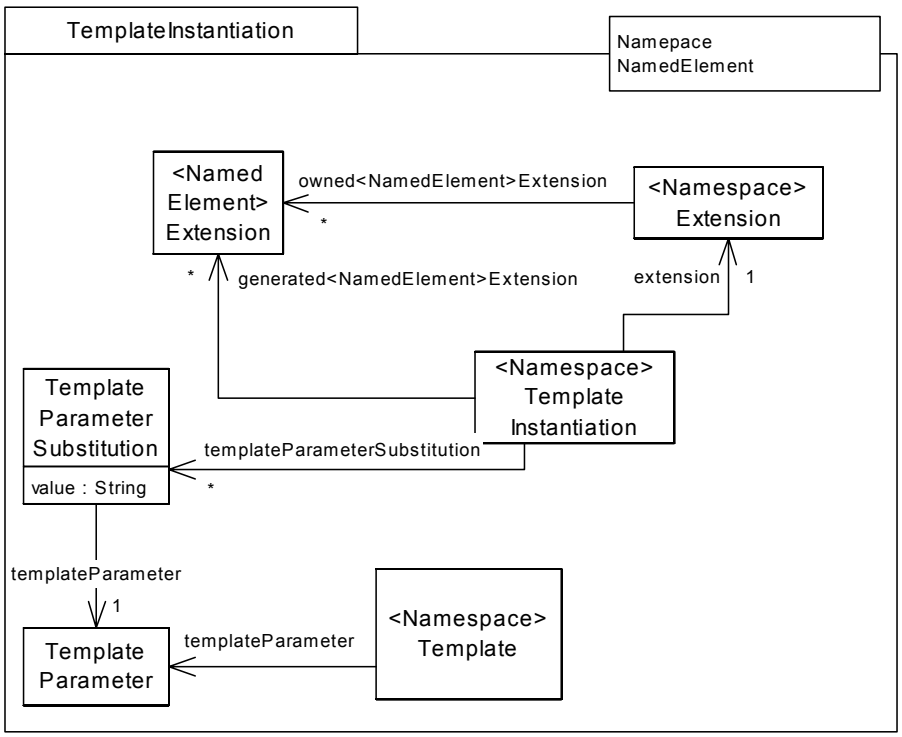
20.14.1 Summary

A general template for defining templateable elements.

20.14.2 Derivation

None.

20.14.3 Definition



<Namespace>Template

A namespace template.

Associations

renamingExpression The renaming expressions that are associated with the contents of the namespace template.

templateParameter The parameters of the namespace template.

<Namespace>TemplateInstantiation

An instantiation relationship.

Associations

templateParameterSubstitution The parameters that are substituted when instantiating the template.

generated<NamedElement>Extension The named element extensions that are generated to realise the instantiation.

20.14.4 Well-formedness Rules

<Namespace>Template

[1] Only one renaming expression per named element in a template.

```
context <Namespace>Template inv:
  self.<namedElement>RenamingExpression -> forAll(r1, r2 |
    r1 <> r2 implies r1.named<NamedElement> <> r2.named<NamedElement>)
```

[2] Only named elements in the template's namespace have renaming expressions associated with them.

```
context <Namespace>Template inv:
  self.member<NamedElement>->
    includesAll(self.<namedElement>RenamingExpression.named<NamedElement>)
```

<Namespace>TemplateInstantiaton

[1] Parameter substitutions parameters must match those owned by the template.

```
context <Namespace>TemplateInstantiation inv:
  self.templateParameterSubstitutions.templateParameter =
  self.ownedParameter->asBag
```

[2] Named element substitutions are generated for each of the renamed named element in the parents namespace.

```
context <Namespace>TemplateInstantiation inv:
  self.generated<namedElement>Extension.parent =
  self.extension.parent.<NamedElement>RenamingExpression.named<NamedElement>
```

[3] Generated named element extensions shadow redefined owned named element extensions.

```
context <Namespace>TemplateInstantiation inv:
  self.extension.owned<NamedElement>Extension->select(e | e.isRedefined) =
  self.generated<NamedElement>Extension
```

[4] The name of the child elements of any generated named element extension is the evaluation of the appropriate renaming expression.

```
context <Namespace>TemplateInstantiation inv:
  self.generated<NamedElement>Extension->forAll(n |
    n.child.name = self.<namedElement>RenamingExpression->
   select(r | r.named<NamedElement> = n.parent).eval(self)->asSet)
```

20.14.5 Operations

Appendix A
Mapping Package to Class Hierarchies

A.1 INTRODUCTION

This appendix gives the rules characterising the mapping between models expressed as a hierarchy of packages related through package extension, and models expressed as a hierarchy of classes. These rules demonstrate that it is possible to produce a class framework suitable to support current approaches to tool construction from a metamodel defined using package extension and package templates.

A.2 OVERVIEW

Source of mapping

A hierarchy of packages (and template packages) related through package extension, where the contents of packages are expressed using packages, classes, class generalisation, attributes, associations, query operations, OCL.

Target of mapping

The subset of the source language including everything but package extension and package templates.

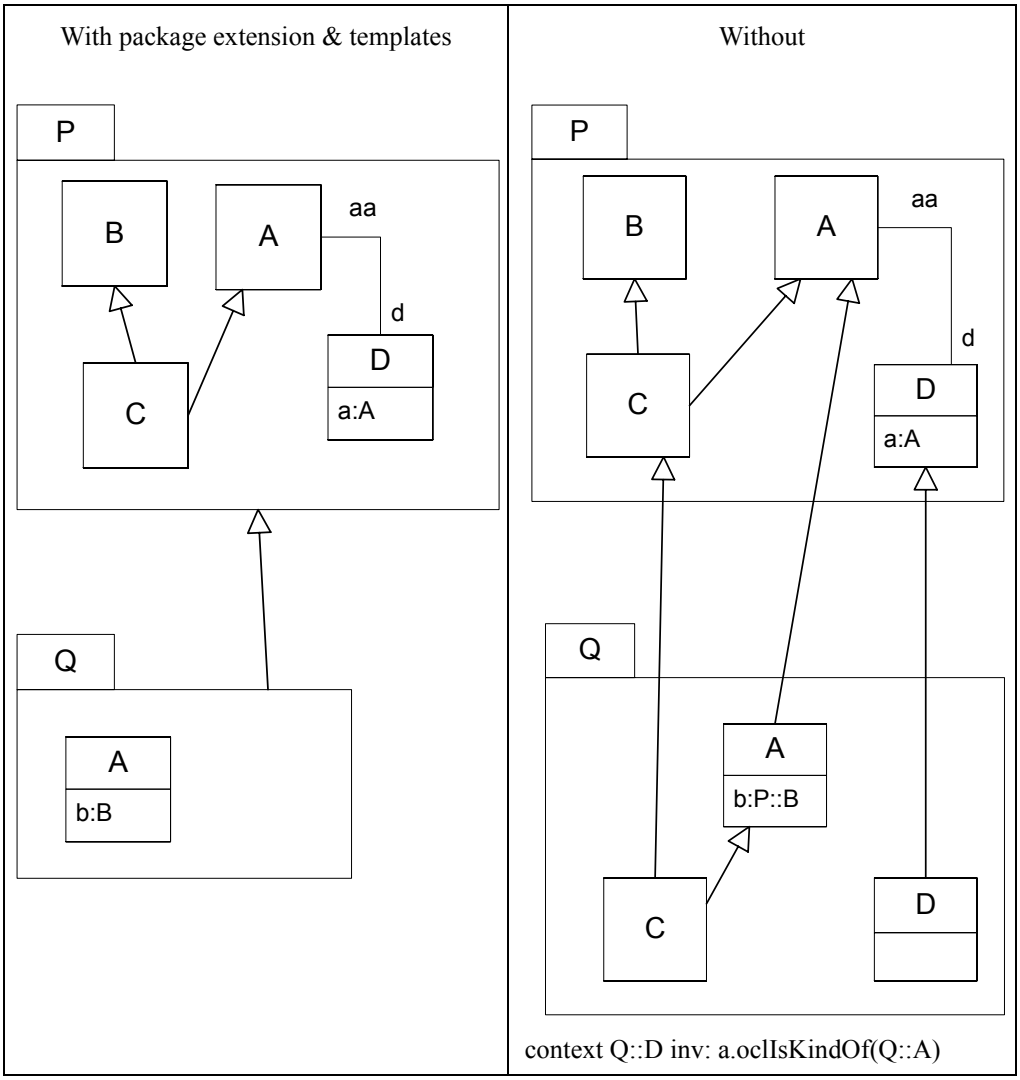
Approach

The eventual goal is to provide a meta-modeled definition of this mapping. If possible, the mapping should be specified so that it is two-way. For this appendix we present the mapping informally on a case by case basis. Short explanations are provided for each case.

We need to consider how various modeling elements within a package get treated in two situations: when there is no renaming on the package extension; when there is renaming on the package extensions. Templates are considered last, as (it turns out) the application of a template is the same as annotating a package extension from that template with renamings generated from the parameters of the template.

A.3 RULES

Classes, Attributes and Associations – no renaming



[a] *B* is not changed in *Q* on LHS, so only *P::B* is required in RHS.

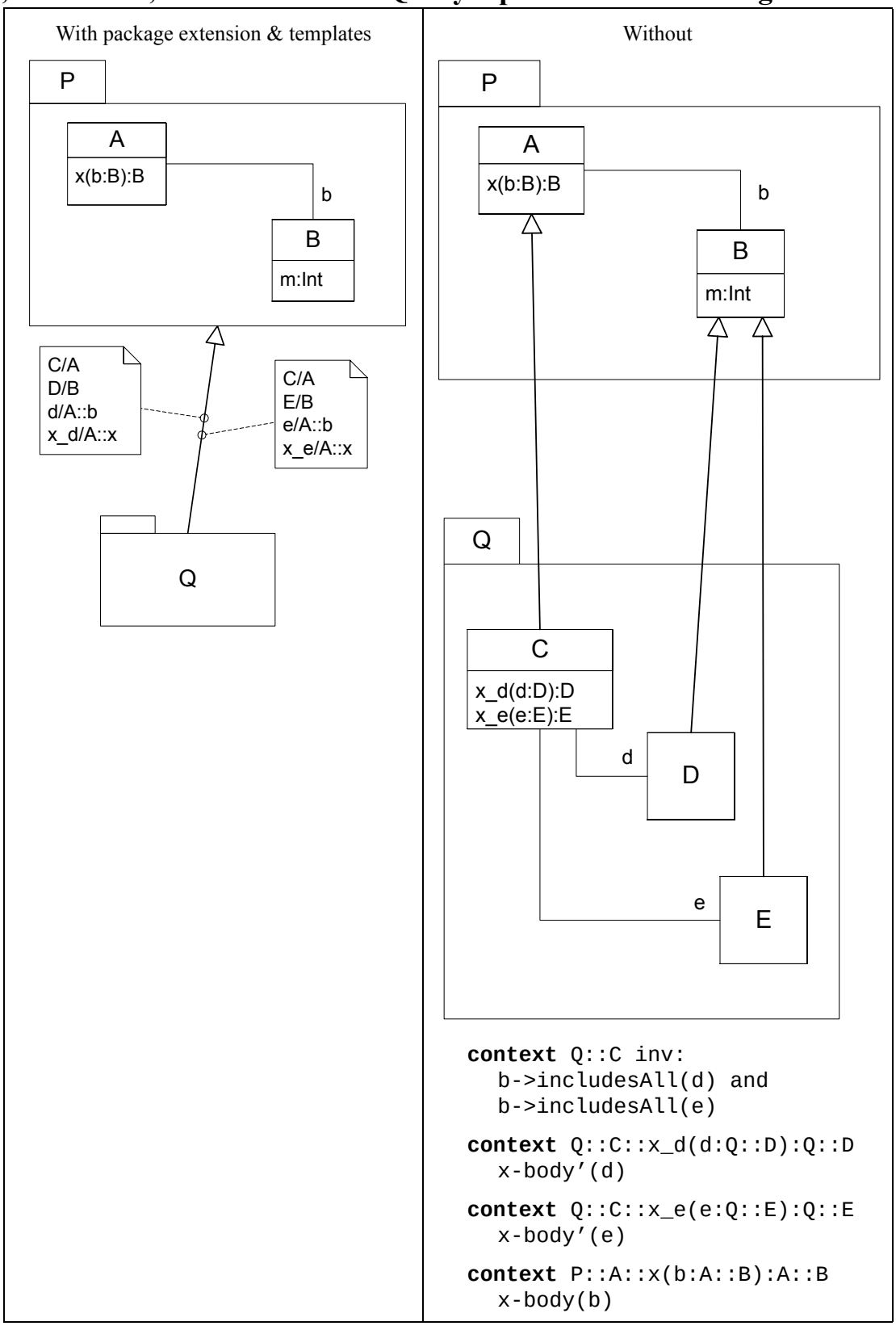
[b] *Q::A* has an attribute added on LHS, so class *Q::A* is required on the RHS. The type of the attribute in class in *Q::A* is *B*, which turns into *P::B* on RHS (see case (a)).

[c] *P::C* inherits from *P::A* on LHS, so depends on *P::A*. By case (b), *A* is changed in *Q* on LHS, requiring *Q::A* on RHS. So need *Q::C* on RHS which inherits from *Q::A* on RHS.

[d] *P::D* has an attribute of type *P::A* on LHS, so depends on *P::A*. By case (b), *A* is changed in *Q* on LHS, so, similar to case (c), class *Q::D* is required on RHS and includes a constraint to strengthen the type of attribute a to *Q::A*.

[e] In a similar way to case (d), association ends of association between *P::A* and *P::D* must be redefined in *Q* on RHS.

Classes, Attributes, Associations and Query Operations – renaming

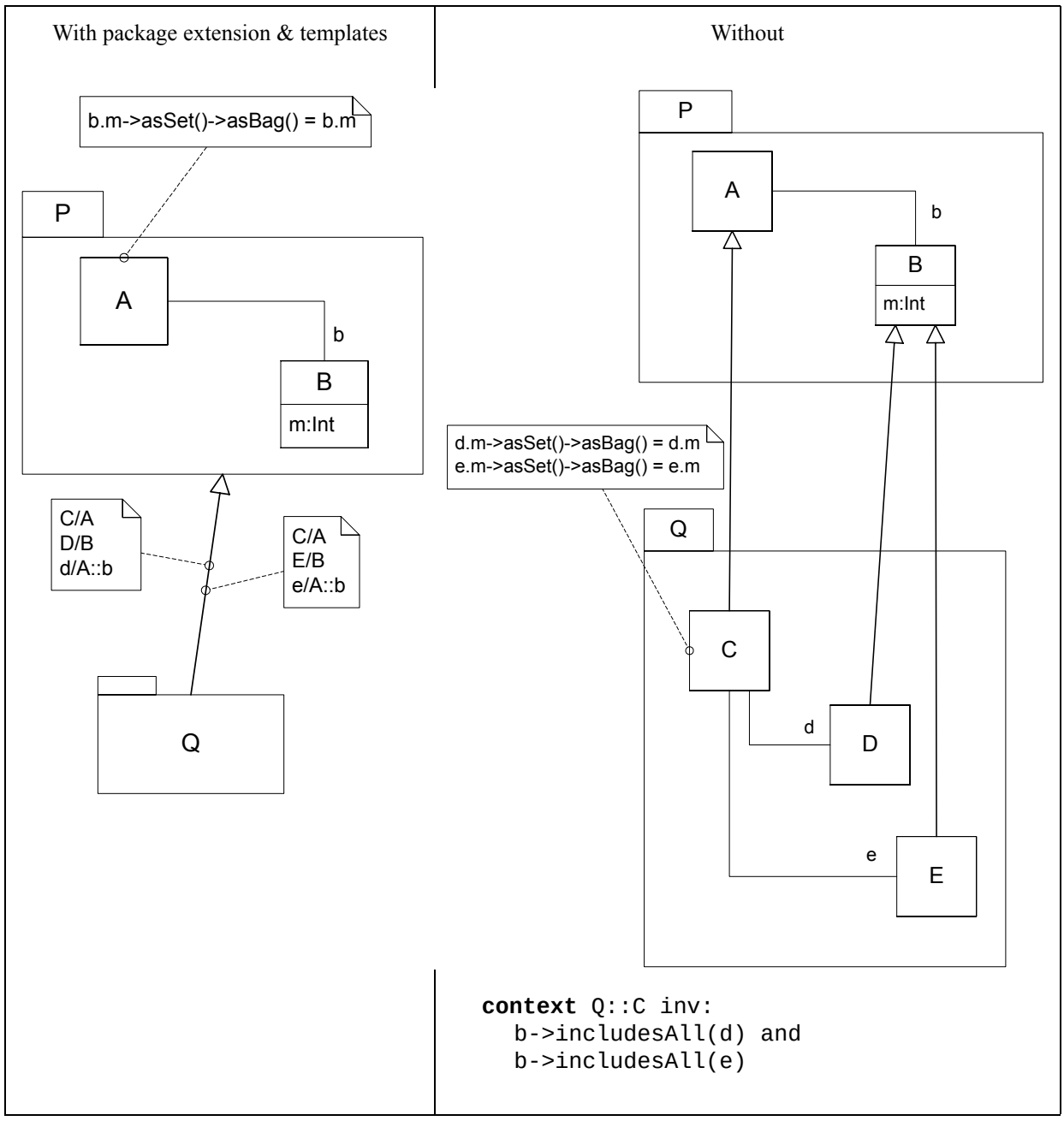


[a] *A* and *B* get renamed when *Q* extends *P* on LHS. Needed matching renamed classes in *Q* on RHS. *A* is renamed to *C* under both extensions of *Q* from *P*, *B* is renamed to *D*, under one extension, and *E*, under the other.

[b] The association end, which also gets renamed twice under package extension, is replicated in *Q* on RHS, once for each new name. Attribute renamings are treated similarly.

[c] Similarly, the query operation in *P::A* gets renamed twice on extension to *Q*. A new operation is introduced in *Q::C* on RHS. *x-body'(d)* is like *x-body* except that all elements from package *P* mentioned in *x-body* are replaced by their renamed counterparts in *Q*, and *b* is replaced by *d*.

Constraints – all cases



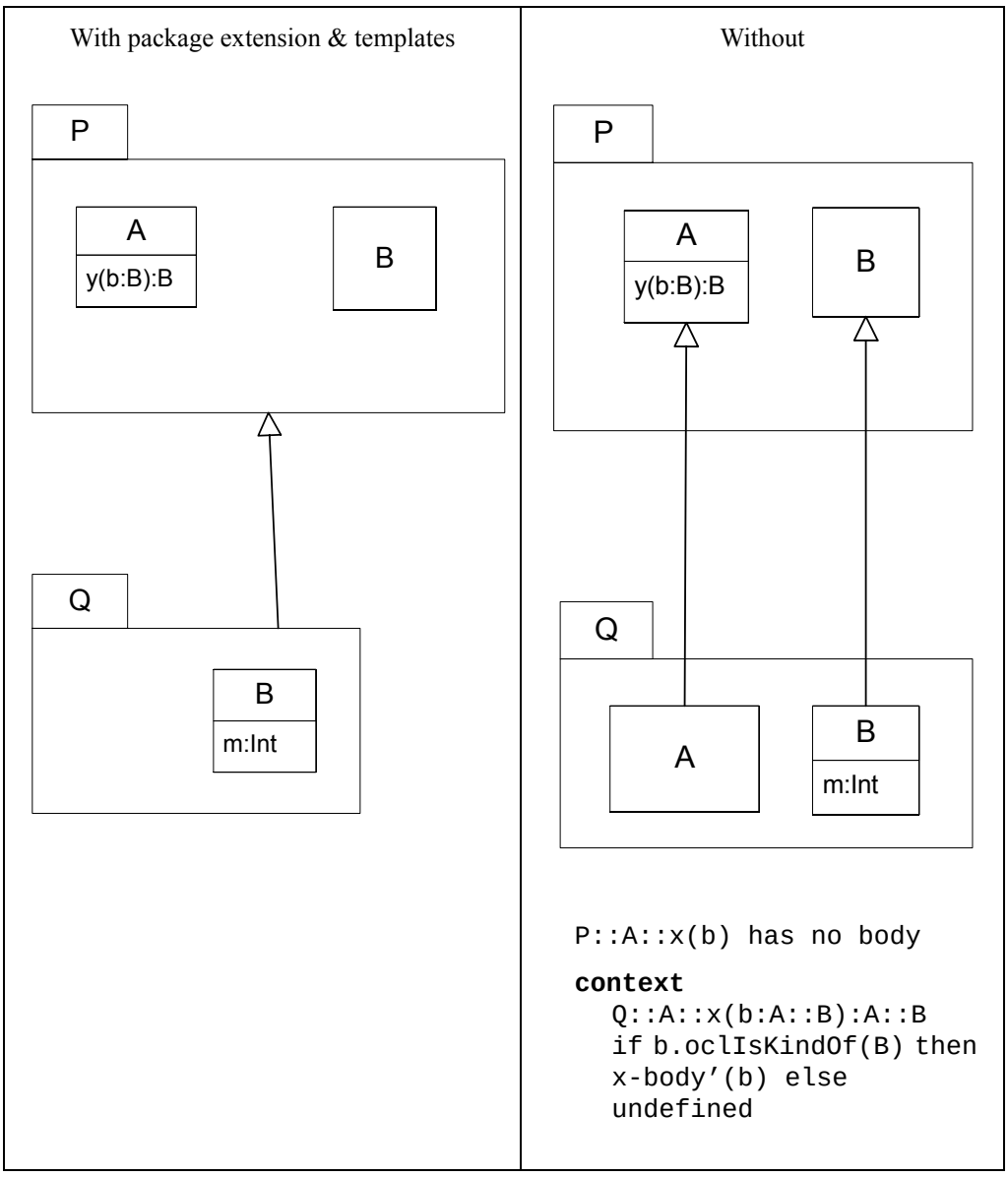
The constraint on *P::A* in LHS gets replicated twice in *Q::C* on RHS, as constraint refers to the association end that gets replicated. The constraint on *P::A* on LHS does not carry over to *P::A* on RHS. This would have the effect of adding an undesirable constraint on *Q::C*, that the union of *d.m* and *e.m* must be unique integers (as *b* includes the union of *d* and *e*), whereas what is actually required is that *d.m* are unique integers and *e.m* are unique integers, with the possibility that there may be integers in *d.m* and *e.m* which are the same.

In fact it is not quite as simple as this. In a hierarchy that is more than two levels deep, constraints should only appear at the lowest level where association ends or attributes involved in the constraint have been replicated.

The only time when a constraint will not be replicated is when it does not involve reference to any attributes, association ends or queries that are renamed by any package extension lower down in the hierarchy which has the package where the constraint is first introduced as a direct or indirect (by transitive traversal of package extensions) parent.

This mapping demonstrates a distinct advantage of the modelling using package extension. It is possible to write constraints on classes in packages that get replicated correcetly in the extension of the package. It is not possible to simulate this using class inheritance, as placing a constraint on the parent class can conflict with the constraints that need to appear lower down.

Query operations – no renaming



B gets changed (an attribute is added) in *Q* on LHS, so by rules in previous section, *Q::B* is required on RHS. *P::A* on LHS has a query *x* that refers to *P::B*, so, as *B* is changed in *Q*, *Q::A* is required on RHS with a body that ensures the appropriate result is returned when the argument is of type *Q::B*, otherwise *undefined* is returned. *x-body'(d)* is like *x-body(b)*, which is the expression defining the query *x* in *P::A* on the LHS, except that all ele-

ments from package *P* mentioned in *x-body(b)* are replaced by their renamed counterparts in *Q*. *P::A::x(b)* has no body; if it did this would conflict with the definition in *Q::A*.

Templates

Templates add to packages is an ability to generate (possibly a large number of) renamings based on the substitution of (possibly a few) template parameters. To do this, a template is associated with a set of parameters, and model elements in the template may be associated with naming expressions. Model elements also have a separate name, which can be used as a useful alias when building the content of the template (e.g. in OCL expressions). By default, the name of the element is the result of evaluating the naming expression with template parameters substituted with their own name. All this means that templates can be treated like normal packages when it comes to a package extension hierarchy, and all the rules above apply.

Bibliography

[UML 1.4] OMG Unified Modeling Language Specification (version 1.4), 2001. Available from www.omg.org.

[MOF 1.3] Meta Object Facility Specification (version 1.3), 2001. Available from www.omg.org.

[OCL 2.0] Response to the UML 2.0 OCL RFP (version 1.4), April 19, 2002. Available from www.klasse.nl.

[Action Semantics] Action Semantics Specification (version 1.4), 2001. Available from www.omg.org,.

[U2Partners] U2 Partners UML 2.0 Draft Submission (version 0.69), 2002. Available from www.u2-partners.org

[Appukuttan02] B.K. Appukuttan, T. Clark, A. Evans, G. Maskeri, P. Sammut, L. Tratt and J. S. Willans. A pattern based approach to defining the dynamic infrastructure of UML 2.0. Presented at the 4th fourth workshop on Rigorous Object Oriented Methods, King's College, March 2002.

[Clark02] A.N.Clark, A.S.Evans, S.Kent. Package Extension and Renaming (<<UML2002>>), Dresden, LNCS, Springer-Verlag, October 2002.

[Clark01a] A.N.Clark, A.S.Evans, S.Kent. A Reference Implementation for UML. In B.Henderson-Sellers and F.Barbier (eds) Object Modelling with UML, Special Issue of L'Objet, Vol 7, no 3/2001, pp363-385, 2001.

[Alvarez01a] J-M Alvarez, A.S.Evans and P.Sammut. Mapping between Levels in the Metamodel Architecture. Proceedings of 4th International Conference on the Unified Modeling Language (<<UML2001>>), Toronto, LNCS 2185, Springer-Verlag, 2001.

[Alverez01b] J-M Alvarez, A.N.Clark, A.S.Evans. An Action Semantics for MML. Proceedings of 4th International Conference on the Unified Modeling Language (<<UML2001>>), Toronto, LNCS 2185, Springer-Verlag, 2001.

[Clark01b] A.N.Clark, A.S.Evans and S.Kent. The Meta-Modeling Language Calculus: Foundation Semantics for UML. Proceedings of FASE Workshop, European Conference of Theory and Practice of Software (ETAPS), Genoa, LNCS, 2001

[Clark01c] A. Clark, A.S.Evans, S. Kent, and P. Sammut. The MMF approach to engineering object-oriented design languages. In Workshop on Language Descriptions, Tools and Applications, LTDA2001, Genoa, 2001.

[Alverez01c] J-M Alvarez, A.S.Evans and P.Sammut. MML and the Meta-Model Architecture. In Workshop on Language Descriptions, Tools and Applications, LTDA2001, Genoa, 2001.

[**Kleppe01]** Anneke Kleppe and Jos Warmer. Unification of Static and Dynamic Semantic in UML. 2001. Available from www.klasse.nl.

[Reggio01] G.Reggio and E. Astesiano. A Proposal for a Dynamic Core for UML Meta-Modelling with MML.Technical Report of DISI - Universit di Genova, DISI-TR-01-17, Italy, 2001. Available from www.disi.unige.it.

[Clark00] A.N.Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. Technical report, pUML Group and IBM, September 2000. Available from www.puml.org.

[Clark99] A.N.Clark, A.S.Evans, R.B.France, S.Kent and B.Rumpe. Response to UML 2.0 Request for Information, December 1999. Available from www.omg.org.

[Evans99] A.S.Evans and S.Kent. Meta-modelling semantics of UML: the pUML approach. 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B.France, Colorado, LNCS 1723, 1999.

[Warmer99] J. Warmer and A. Kleppe. The Object Constraint Language: precise modeling with UML, Addison Wesley, 1999.

[D'Souza98] D. D'Souza and A.Wills. Object, Components and Frameworks with UML: The Catalysis Approach. Addison Wesley, 1998.

[Griffiths97] A.Griffiths. Object-oriented Operations have Two Parts. University of Queensland. Technical Report 97-20, Australia, 1997.

[Lamport89] A Simple Approach to Specifying Concurrent Systems. Communications of the ACM, 32(1):32-45, 1989.

[Chandy88] Parallel Program Design: A Foundation. Addison Wesley, 1998.